
**THE
WANG
PROFESSIONAL
COMPUTER**

Pascal
Reference
Manual

WANG

The Wang Professional Computer Pascal Reference Manual

1st Edition — July, 1983

Copyright © Wang Laboratories, Inc., 1983

Portions reproduced with permission of Microsoft, Inc.

700-8321

WANG

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL. 617/459-5000 TWX 710 343-6769 TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase, lease, or license agreement by which this software package was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of the software package, the accompanying manual, or any related materials.

NOTICE:

All Wang Program Products are licensed to customers in accordance with the terms and conditions of the Wang Laboratories, Inc. Standard Program Products License; no ownership of Wang Software is transferred and any use beyond the terms of the aforesaid License, without the written authorization of Wang Laboratories, Inc., is prohibited.

**WANG**

WANG LABORATORIES, INC., ONE INDUSTRIAL AVENUE, LOWELL, MA 01851 • TEL. 617/459-5000, TWX 710-343-6769, TELEX 94-7421

PREFACE

The purpose of the Wang Professional Computer Pascal Reference Manual is to acquaint you with the Wang Professional Computer Pascal compiler and the Pascal language as implemented on the Wang Professional Computer (PC). This manual assumes that you know how to program in Pascal. If you are unfamiliar with Pascal, refer to Section 1.4 for a list of texts that will help you learn to program in Pascal.

This manual consists of two main parts. The first part, consisting of Chapters 1 through 9, explains how to use the Wang Professional Computer Pascal compiler. This part includes general information on the compiling process, a demonstration run in which you compile, link, and execute a Wang PC Pascal sample program, and more detailed discussions of each phase of the compiling process. Read this part first before you attempt to compile your own Pascal source programs. The second part, which includes Chapters 10 through 26, explains Wang PC Pascal syntax in detail. You need not read this part in its entirety before compiling your own programs. Part Two is a reference you can consult for more specific information on a particular component of Wang PC Pascal syntax.

Use this manual in conjunction with The Wang Professional Computer Program Development Guide (700-8018) and The Wang Professional Computer Introductory Guide (700-8020).

CONTENTS

PART ONE	USING THE WANG PC PASCAL COMPILER	
CHAPTER 1	INTRODUCTION	
1.1	System Requirements	1-1
1.2	Diskette Contents	1-2
1.3	Notation Used in Chapters 1 through 9	1-3
1.4	Resources for Learning Pascal	1-3
CHAPTER 2	GETTING STARTED	
2.1	Preliminary Procedures	2-1
	Backing Up Your System Files	2-1
	Copying PASKEY to the Default Drive	2-1
	Setting Up Your System Disks	2-1
2.2	Program Development	2-2
CHAPTER 3	DEMONSTRATION RUN	
3.1	Introduction	3-1
3.2	Demonstration Run	3-1
CHAPTER 4	COMPILING	
4.1	Files the Compiler Writes	4-1
	The Object File	4-1
	The Source Listing File	4-1
	The Object Listing File	4-2
	The Intermediate Files	4-2
4.2	File Name Conventions	4-3
4.3	Entering the Pascal Compiler	4-6
	Specifying No Command Line Parameters	4-6
	Specifying All Command Line Parameters	4-7
	Specifying Some Command Line Parameters	4-8
4.4	Pass 1 Compiler Switches	4-8
CHAPTER 5	LINKING	
5.1	Files the Linker Reads	5-1
	Object Modules	5-1
	Libraries	5-3

CONTENTS (continued)

5.2	Files the Linker Writes	5-4
	The Run File	5-4
	The Linker Listing File	5-4
	VM.TMP	5-5
5.3	Linker Switches	5-5
CHAPTER 6	USING A BATCH COMMAND FILE	
CHAPTER 7	COMPILING AND LINKING LARGE PROGRAMS	
7.1	Limits on Code Size	7-1
7.2	Limits on Data Size	7-1
7.3	Working With Limits on Compiler Memory	7-3
	Identifiers	7-3
	Complex Expressions	7-4
7.4	Working With Limits on Disk Memory	7-5
	Pass 1	7-5
	Pass 2	7-6
	Linking	7-7
	A Complex Example	7-8
7.5	Minimizing Load Module Size	7-9
	I/O	7-9
	Runtime Error Handling	7-10
	Real Number Operations	7-10
	Error Checking	7-10
CHAPTER 8	USING ASSEMBLY LANGUAGE ROUTINES	
8.1	Calling Conventions	8-1
8.2	Internal Representations of Data Types	8-3
8.3	Interfacing to Assembly Language Routines	8-6
CHAPTER 9	ADVANCED TOPICS	
9.1	The Structure of the Compiler	9-1
	The Front End	9-2
	The Back End	9-3
9.2	An Overview of the File System	9-5
9.3	Runtime Architecture	9-8
	Runtime Routines	9-8
	Memory Organization	9-9
	Initialization and Termination	9-11
	Error Handling	9-16

CONTENTS (continued)

PART TWO WANG PC PASCAL SYNTAX

CHAPTER 10 LANGUAGE OVERVIEW

10.1	Metacommmands	10-1
10.2	Programs and Compilable Parts of Programs	10-1
10.3	Procedures and Functions	10-4
10.4	Statements	10-5
10.5	Expressions	10-6
10.6	Variables	10-7
10.7	Constants	10-8
10.8	Types	10-9
10.9	Identifiers	10-10
10.10	Notation	10-11

CHAPTER 11 PASCAL NOTATION

11.1	Components of Identifiers	11-1
	Letters	11-1
	Digits	11-2
	The Underscore Character	11-2
11.2	Separators	11-2
11.3	Special Symbols	11-3
	Punctuation	11-3
	Operators	11-3
	Reserved Words	11-5
11.4	Unused Characters	11-5

CHAPTER 12 IDENTIFIERS

12.1	Introduction	12-1
12.2	Declaring an Identifier	12-2
12.3	The Scope of Identifiers	12-3
12.4	Predeclared Identifiers	12-4

CHAPTER 13 INTRODUCTION TO DATA TYPES

13.1	Understanding Types	13-1
13.2	Declaring Data Types	13-1
13.3	Type Compatibility	13-3
	Type Identity and Reference Parameters	13-3
	Type Compatibility and Expressions	13-4
	Assignment Compatibility	13-5

CONTENTS (continued)

CHAPTER 14 SIMPLE TYPES

14.1	Ordinal Types	14-1
	INTEGER	14-1
	WORD	14-2
	CHAR	14-2
	BOOLEAN	14-3
	Enumerated Types	14-3
	Subrange Types	14-4
14.2	REAL	14-5
14.3	INTEGER4	14-7

CHAPTER 15 ARRAYS, RECORDS, AND SETS

15.1	Arrays	15-1
15.2	Super Arrays	15-2
	STRINGS	15-5
	LSTRINGS	15-6
	Using STRINGS and LSTRINGS	15-7
15.3	Records	15-11
	Variant Records	15-12
	Explicit Field Offsets	15-14
15.4	Sets	15-15

CHAPTER 16 FILES

16.1	Declaring Files	16-1
16.2	The Buffer Variable	16-2
16.3	File Structures	16-3
	BINARY Structure Files	16-3
	ASCII Structure Files	16-4
16.4	File Access Modes	16-4
	TERMINAL Mode Files	16-5
	SEQUENTIAL Mode Files	16-5
	DIRECT Mode Files	16-5
16.5	The Predeclared Files INPUT and OUTPUT	16-6
16.6	Extend Level I/O	16-7
16.7	System Level I/O	16-8

CHAPTER 17 REFERENCE AND OTHER TYPES

17.1	Reference Types	17-1
	Pointer Types	17-1
	Address Types	17-3
	Segment Parameters for the Address Types	17-6
	Using the Address Types	17-7
	Notes on Reference Types	17-8

CONTENTS (continued)

17.2	PACKED Types	17-8
17.3	Procedural and Functional Types	17-9
CHAPTER 18	CONSTANTS	
18.1	Introduction	18-1
18.2	Declaring Constant Identifiers	18-2
18.3	Numeric Constants	18-2
	REAL Constants	18-3
	INTEGER, WORD, and INTEGER4 Constants	18-4
	Nondecimal Numbering	18-5
18.4	Character Strings	18-6
18.5	Structured Constants	18-7
18.6	Constant Expressions	18-9
CHAPTER 19	VARIABLES AND VALUES	
19.1	Introduction	19-1
19.2	Declaring a Variable	19-2
19.3	The Value Section	19-2
19.4	Using Variables and Values	19-3
	Components of Entire Variables and Values	19-4
	Reference Variables	19-6
19.5	Attributes	19-7
	The STATIC Attribute	19-8
	The PUBLIC and EXTERN Attributes	19-9
	The ORIGIN and PORT Attribute	19-10
	The READONLY Attribute	19-10
	Combining Attributes	19-11
CHAPTER 20	EXPRESSIONS	
20.1	Simple Type Expressions	20-2
20.2	Boolean Expressions	20-5
20.3	Set Expressions	20-7
20.4	Function Designators	20-9
20.5	Evaluating Expressions	20-10
20.6	Other Features of Expressions	20-11
	The EVAL Procedure	20-11
	The RESULT Function	20-12
	The RETYPE Function	20-12
CHAPTER 21	STATEMENTS	
21.1	The Syntax of Pascal Statements	21-1
	Labels	21-1
	Separating Statements	21-2

CONTENTS (continued)

	The Reserved Words BEGIN and END	21-3
21.2	Simple Statements	21-3
	Assignment Statements	21-4
	Procedure Statements	21-5
	The GOTO Statement	21-6
	The BREAK, CYCLE, and RETURN Statements	21-8
21.3	Structured Statements	21-8
	Compound Statements	21-9
	Conditional Statements	21-10
	Repetition Statements	21-12
	The WITH Statement	21-16
	Sequential Control	21-16

CHAPTER 22 INTRODUCTION TO PROCEDURES AND FUNCTIONS

22.1	Procedures	22-2
22.2	Functions	22-3
22.3	Attributes and Directives	22-4
	The FORWARD Directive	22-7
	The EXTERN Directive	22-7
	The PUBLIC Attribute	22-8
	The ORIGIN Attribute	22-9
	The FORTRAN Attribute	22-9
	The INTERRUPT Attribute	22-10
	The PURE Attribute	22-11
22.4	Procedure and Function Parameters	22-12
	Value Parameters	22-13
	Reference Parameters	22-13
	Procedural and Functional Parameters	22-16

CHAPTER 23 AVAILABLE PROCEDURES AND FUNCTIONS

23.1	Categories of Available Procedures and Functions	23-1
	File System Procedures and Functions	23-2
	Dynamic Allocation Procedures	23-3
	Data Conversion Procedures and Functions	23-3
	Arithmetic Functions	23-4
	Extend Level Intrinsic Procedures	23-6
	System Level Intrinsic Procedures	23-6
	String Intrinsic Procedures	23-6
	Library Procedures and Functions	23-7
23.2	Directory of Functions and Procedures	23-8

CONTENTS (continued)

CHAPTER 24 FILE-ORIENTED PROCEDURES AND FUNCTIONS

24.1	File System Primitive Procedures and Functions	24-2
	GET and PUT	24-2
	RESET and REWRITE	24-3
	EOF and EOLN	24-4
	PAGE	24-5
	Lazy Evaluation	24-5
	Concurrent I/O	24-7
24.2	Textfile Input and Output	24-8
	READ and READLN	24-10
	READ Formats	24-12
	WRITE and WRITELN	24-14
	WRITE Formats	24-15
24.3	Extend Level I/O	24-17
	Extend Level Procedures	24-18
	Temporary Files	24-21

CHAPTER 25 COMPILABLE PARTS OF A PROGRAM

25.1	Programs	25-2
25.2	Modules	25-5
25.3	Units	25-7
	The Interface Division	25-11
	The Implementation Division	25-12

CHAPTER 26 WANG PC PASCAL METACOMMANDS

26.1	Language Level Setting and Optimization	26-2
26.2	Debugging and Error Handling	26-4
26.3	Source File Control	26-9
26.4	Listing File Control	26-11
26.5	Listing File Format	26-15
26.6	Command Line Switches	26-17

APPENDICES

Appendix A	Version Specifics	A-1
Appendix B	Wang PC Pascal Features and the ISO Standard	B-1
Appendix C	Wang PC Pascal and Other Pascals	C-1
Appendix D	ASCII Character Codes	D-1
Appendix E	Summary of Wang PC Pascal Reserved Words	E-1
Appendix F	Summary of Available Procedures and Functions	F-1
Appendix G	Summary of Wang PC Pascal Metacommands	G-1
Appendix H	Error Messages	H-1

CONTENTS (continued)

GLOSSARY Glossary-1

INDEX Index-1

FIGURES

Figure 2-1	Program Development	2-5
Figure 8-1	Contents of the Frame	8-1
Figure 8-2	Stack Before Transfer to ADD	8-7
Figure 8-3	1-Byte Return Value	8-8
Figure 8-4	2-Byte Return Value	8-8
Figure 8-5	4-Byte Return Value	8-9
Figure 9-1	The Structure of the Wang PC Pascal Compiler	9-1
Figure 9-2	The Unit U Interface	9-7
Figure 9-3	Memory Organization	9-10
Figure 9-4	Wang PC Pascal Program Structure	9-13
Figure 16-1	Buffer Variable and File	16-2
Figure 25-1	A Wang PC Pascal Unit	25-7
Figure 25-2	Unit with File X.INT	25-8
Figure 26-1	Listing File Format	26-15

TABLES

Table 2-1	A Suggested Disk Setup	2-2
Table 4-1	Conventional File Name Extensions	4-4
Table 4-2	File Names Assigned by the Compiler	4-4
Table 4-3	Pass 1 Compiler Switches	4-9
Table 5-1	Linker Defaults	5-3
Table 9-1	Unit Identifier Suffixes	9-8
Table 9-2	Error Code Classification	9-17
Table 9-3	Runtime Values in BRTEQQ	9-18
Table 10-1	Summary of Wang PC Pascal Statements	10-6
Table 11-1	Summary of Punctuation in Wang PC Pascal	11-4
Table 12-1	Declaring Identifiers	12-3
Table 13-1	Categories of Types in Wang PC Pascal	13-2
Table 17-1	Relative and Segmented Machine Addresses	17-4
Table 18-1	INTEGER, WORD, and INTEGER4 Constants	18-4
Table 18-2	Constant Conversions	18-5
Table 18-3	Constant Operators and Functions	18-9
Table 19-1	Attributes for Variables	19-7
Table 20-1	Expressions	20-1
Table 20-2	Set Operators	20-7
Table 21-1	Wang PC Pascal Statements	21-1
Table 22-1	Attributes and Directives for Procedures and Functions	22-5
Table 23-1	Categories of Available Procedures and Functions	23-2
Table 23-2	File System Procedures and Functions	23-3
Table 23-3	Predeclared Arithmetic Functions	23-5
Table 23-4	Functions from the Wang PC FORTRAN Runtime Library	23-5
Table 23-5	String Procedures and Functions	23-7
Table 23-6	Conversion to INTEGER	23-24

TABLES (continued)

Table 23-7	Conversion to WORD	23-33
Table 24-1	File System Procedures and Functions	24-1
Table 24-2	Lazy Evaluation	24-6
Table 26-1	Metacommand Notation	26-2
Table 26-2	Language and Optimization Level	26-3
Table 26-3	Debugging and Error Handling	26-4
Table 26-4	Source File Control	26-9
Table 26-5	Listing File Control Metacommands	26-12
Table 26-6	Symbol Table Notation	26-14
Table 26-7	Command Line Switches	26-18
Table C-1	Wang PC Pascal and UCSD Pascal	C-3
Table F-1	Procedures and Functions	F-1
Table G-1	Wang PC Pascal Metacommands	G-1

Part 1

Using the
Wang Professional Computer
Pascal Compiler

CHAPTER 1

INTRODUCTION

The Wang Professional Computer (PC) Pascal compiler accepts programs written according to International Standards Organization (ISO) standards. It also accepts programs written in the Wang PC Pascal language, as described in Part Two (Chapters 10 through 26) of this manual.

1.1 SYSTEM REQUIREMENTS

In addition to the minimum configuration, your Wang Professional Computer must have the following items to run the Wang PC Pascal compiler:

- You must install a Memory Expansion card in your electronics unit before you can run Wang PC Pascal, as the compiler requires more than the standard 128 KB of memory. Refer to The Wang Professional Computer Introductory Guide for more information on this card.
- In addition to the files on the Wang PC Pascal diskettes, you need a text editor to create source programs (the Program Development diskette contains the Wang PC Editor, PCEDIT.EXE); and the Wang PC Linker, LINK.EXE, to link your program to modules in the runtime library. Refer to The Wang Professional Computer Program Development Guide for more information about these files.

1.2 DISKETTE CONTENTS

The Wang PC Pascal files are on three dual-sided double-density (DSDD) diskettes. These diskettes and their contents are:

- Disk 1:

<u>File</u>	<u>Function</u>
PAS1.EEM	Pass 1 of the compiler -- parsing
PAS2.EXE	Pass 2 of the compiler -- code generation and optimization
PASKEY	A data file the compiler uses during Pass 1
SORT.PAS and PRIMES.PAS	Demonstration programs
FINU, FINK, and FINKOM	Low level file handling declarations

Before you can use the Wang PC Pascal Compiler, you must change the name of the file PAS1.EEM to PAS1.EXE with the FILE RENAME utility on the System Utilities Menu. Refer to The Wang Professional Computer Introductory Guide for instructions on how to use the FILE RENAME utility.

- Disk 2:

<u>File</u>	<u>Function</u>
PASCAL.LEM	The Pascal runtime library
PAS1.E87	Pass 1 of the compiler for systems equipped with an 8087 coprocessor
PASCAL.L87	Runtime library for systems equipped with an 8087 coprocessor
PASCAL.P87	Library map of PASCAL.L87
PASCAL.PEM	Library map of PASCAL.LIB
NULF.OBJ	Dummy file system
NULE6.OBJ	Dummy error system
NULR7.OBJ	Dummy real number system
NULL.LIB	An empty library

Before you can use the Wang PC Pascal Compiler, you must change the name of the file PASCAL.LEM to PASCAL.LIB with the FILE RENAME utility on the System Utilities Menu. Refer to The Wang Professional Computer Introductory Guide for instructions on how to use the FILE RENAME utility.

NOTE:

The current implementation of the Wang PC does not support the 8087 numeric coprocessor. The 8087 coprocessor will, however, be available for use with future versions of the Wang PC.

- Tools Disk

<u>File</u>	<u>Function</u>
PAS3.EXE	Pass three of the compiler (disassembler)
ENTXL6L.ASM	Assembler source for initialization and termination routines

The tools disk also contains several assorted tool and utility files.

1.3 NOTATION USED IN CHAPTERS 1 THROUGH 9

Most punctuation marks and special characters used in command formats are part of the command format. You must use them as they appear in the command. Some characters used in command formats, however, have special meanings:

- Capital Letters (FOO) -- Indicate that you must enter the parameter or command exactly as shown.
- Angle Brackets (<>) -- Indicate that the text they enclose specifies a class of parameters. Any parameter that you enter in this position must be a valid member of that parameter class. For example, <filespec> means that you must enter a legal filespec. Capital letters enclosed by angle brackets specify nondisplayable WISSCII (a superset of ASCII) characters.
- Square Brackets ([]) -- Indicate that the enclosed parameter is optional. For instance, <filespec>[.<filespec>] specifies entry of either one filespec or two filespecs.
- Ellipses (...) -- Indicate that you can enter the language element preceding the ellipses as many times as needed. For example, <filespec>... indicates entry of one or more filespecs.

1.4 RESOURCES FOR LEARNING PASCAL

This manual provides complete instructions for using the Pascal compiler. The Wang Professional Computer set of manuals does not include any instructional material for the Pascal language. The Wang Professional Computer Pascal Reference Manual is strictly a syntax and semantics reference for Pascal as implemented on the Wang Professional Computer. To learn to program in Pascal, you can use the following texts:

- Findlay, W., and Watt, D. F. Pascal: An Introduction to Methodical Programming. Pittman: London, 1978.
- Holt, Richard C., and Hume, J. N. P. Programming Standard Pascal. Reston Publishing Company: Reston, Va., 1980.

- Jensen, Kathleen, and Wirth, Niklaus. Pascal User Manual and Report. Springer-Verlag: New York, 1974, 1978.
- Koffman, E. B. Problem Solving and Structured Programming in Pascal. Addison-Wesley Publishing Company: Reading, Mass., 1981.
- Schneider, G. M., Weinhart, S. W., and Perlman, D. M. An Introduction to Programming and Problem Solving With Pascal. John Wiley & Sons: New York, second edition, 1982.

CHAPTER 2 GETTING STARTED

2.1 PRELIMINARY PROCEDURES

This section describes several preliminary procedures you should perform before you begin the sample session or compile any programs of your own. If you are unfamiliar with any of these procedures, consult The Wang Professional Computer Introductory Guide.

2.1.1 Backing Up Your System Files

The first thing you should do after you unwrap your Wang PC Pascal diskettes is to make copies to work with, saving the original diskettes for backup. Make the copies using the DISK COPY utility supplied with your System Software diskettes. Refer to The Wang Professional Computer Introductory Guide for information on the DISK COPY utility.

2.1.2 Copying PASKEY to the Default Drive

This step is required. PASKEY, one of the files that comes as a part of the Wang PC Pascal Compiler, contains the Wang PC Pascal predeclarations. Because Pass 1 uses these predeclarations, the PASKEY file must always be on the disk in the default drive while Pass 1 is executing. Before you begin to compile the sample program or any program of your own, copy PASKEY onto the disk in the default drive.

2.1.3 Setting Up Your System Disks

Before you begin compiling and linking a program, check the contents of each diskette. You may wish to copy some files from one system disk to another to set up a working arrangement that is convenient for you. You will certainly need to have the Linker available (from your Program Development Tools diskette).

In order to avoid continual reprompting from the system to reload the system files, you may wish to set up your diskettes as shown in Table 2-1. This setup assumes you have two 320 or 360 KB disk drives available.

Table 2-1. A Suggested Diskette Setup

Diskette Number	Contents
1	COMMAND.COM and other system files PCEDIT.EXE PRIMES.PAS SORT.PAS PAS1.EXE PASKEY Your Pascal source files
2	PASCAL.LIB LINK.EXE PAS2.EXE PAS3.EXE

NOTE:

If you have not yet changed the name of the file PAS1.EEM to PAS1.EXE and changed the name of the file PASCAL.LEM to PASCAL.LIB, you must do so before you can use the compiler. Refer to The Wang Professional Computer Introductory Guide for instructions on how to use the FILE RENAME utility.

You can copy COMMAND.COM and the system files to diskette 1 at the time you format the diskette. Refer to The Wang Professional Computer Introductory Guide for formatting procedures.

2.2 PROGRAM DEVELOPMENT

This section provides a short introduction to program development, a multistep process that includes writing the program, and compiling, linking, and executing it. For a brief explanation of terms that may be unfamiliar, refer to the Glossary in the back of this manual.

A microprocessor can execute only its own machine instructions; it cannot execute source program statements directly. Before you can run a program, the microprocessor must translate the statements in your program to machine language. Compilers and interpreters are two types of programs that perform this translation. Depending on the language you are using, either or both types of translation may be available to you. Wang PC Pascal is a compiled language.

A compiler translates a source program and creates a new file called an object file. The object file contains relocatable machine code that the compiler can place and run at different absolute locations in memory.

Compilation also associates memory addresses with variables and with the targets of GOTO statements, so that the compiler need not search lists of variables or of labels during execution of your program.

Getting Started

Many compilers, including the Wang PC Pascal Compiler, are "optimizing" compilers. During optimization, the compiler reorders expressions and eliminates common subexpressions, both to increase execution speed and to decrease program size. These factors measurably increase the execution speed of your program.

The Wang PC Pascal Compiler has a three-part structure. The first two parts, Pass 1 and Pass 2, carry out the optimization and create the object code. Pass 3 is an optional step that creates an object code listing. Chapter 4 describes compiling in greater detail.

Before you can execute a successfully compiled program, you must link it to modules in the runtime library. Linking is the process in which the Linker computes absolute offset addresses for routines and variables in relocatable object modules and then resolves all external references by searching the runtime library. The Linker saves your program on disk as an executable file, ready to run.

At link time, you can link more than one object module, as well as routines written in assembly language or other high-level languages and routines in other libraries. Chapter 5 describes linking in greater detail.

The program development process consists of the following steps:

1. Create and edit the Wang PC Pascal (and Wang PC Assembly) source file. Program development begins when you write a Wang PC Pascal program. Any general-purpose text editor will serve, but your most readily available choice is the Wang PC Editor, PCEDIT.EXE. You must also use a text editor to write any assembly language routines you plan to include.
2. Compile the program using the \$DEBUG+ metacommand (refer to Chapter 26). Assemble the assembly source, if any. Once you have written a program, compile it with the Wang PC Pascal Compiler. The compiler flags all syntax and logic errors as it reads your source file. Use the error-checking switches or their corresponding metacommands (described in Section 3.4) to generate diagnostic calls for all runtime errors. If compilation is successful, the compiler creates a relocatable object file.

If you have written your own assembly language routines (for example, to increase the speed of execution of a particular algorithm), assemble those routines with Wang PC Assembly.

3. Link compiled (and assembled) .OBJ files with the runtime library. You must link your compiled (or assembled) object file with one of the runtime libraries, using the Linker. You can also link separately-compiled Wang PC FORTRAN subroutines to your program at this time.
4. Run the .EXE file. The Linker links all modules your program needs and produces an executable object file with .EXE as the extension. Type the file name to execute this file.

5. Recompile, relink, and rerun with \$DEBUG-. Repeat this process until your program successfully compiles, links, and runs without errors. Then recompile, relink, and rerun it without the runtime error-checking switches to reduce the amount of time and space required. Chapter 6 discusses how to work within various physical limitations you can encounter in compiling, linking, and executing a program.

Figure 2-1 illustrates the entire program development process.

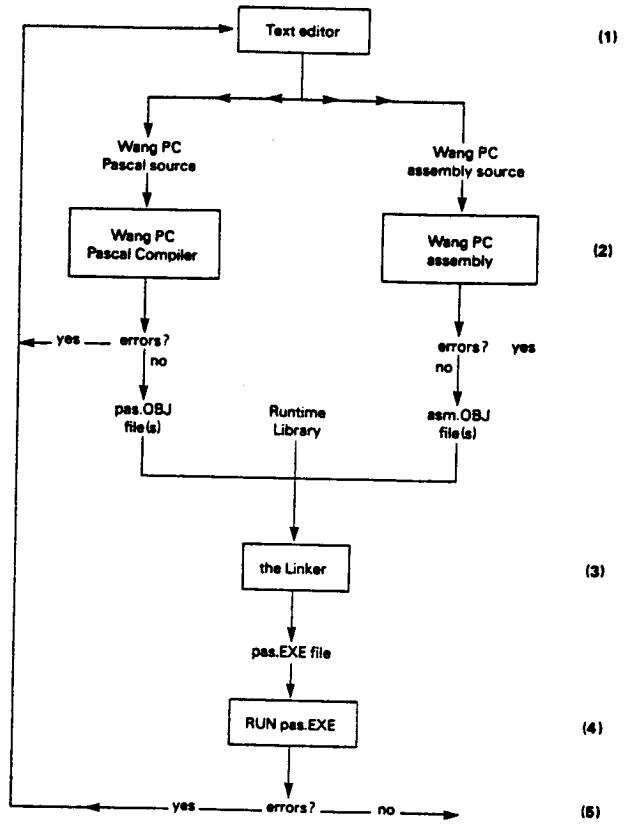


Figure 2-1. Program Development

CHAPTER 3

DEMONSTRATION RUN

3.1 INTRODUCTION

This chapter contains a demonstration run that tells you how to compile, link, and execute the Wang PC Pascal demonstration program PRIMES.PAS. Compile this demonstration program before you attempt to compile your own Pascal programs. The demonstration run assumes that your Wang Professional Computer has two diskette drives, and that your diskettes are set up as shown in Table 2-1.

3.2 DEMONSTRATION RUN

The steps in developing a program with the Wang PC Pascal compiler are:

- Editing (enter and correct the Pascal program)
- Compiling (create a relocatable object file — this occurs in three stages called passes)
- Linking (create an executable program)
- Running (execute the program)

In the following discussion, all text that you enter is underlined. You must press RETURN at the end of each line you enter. When the phrase (press RETURN) appears in an example, press RETURN only — do not enter any text. To create an executable compiled program, perform the following steps.

NOTE:

If you have not yet changed the name of the file PAS1.EEM to PAS1.EXE and changed the name of the file PASCAL.LEM to PASCAL.LIB, you must do so before you can use the compiler. Refer to The Wang Professional Computer Introductory Guide for instructions on how to use the FILE RENAME utility.

STEP 1 — Power on your system

Refer to The Wang Professional Computer Introductory Guide for power-on procedures.

STEP 2 -- Create a Pascal source file

You can create Pascal programs with the Wang PC Editor, PCEDIT.EXE. Refer to The Wang Professional Computer Program Development Guide for information on how to use the Editor. For this demonstration run, however, use the program PRIMES.PAS on your Diskette 1. For consistency, always give Pascal source files the .PAS extension; the operating system gives Pascal source files the .PAS extension by default.

STEP 3 -- Enter the compiler

Choose the DOS Command Processor option from the Main System Menu. When the A: prompt appears on the screen, insert Diskette 1 in Drive A and insert Diskette 2 in Drive B. Respond to the A: prompt by entering:

A:PAS1

When you enter this command, you leave the operating system and enter the Pascal compiler. The compiler displays a message similar to the following one:

Microsoft MS-Pascal Compiler, MS-DOS Test Version 3.01, 10/82

STEP 4 -- Enter the file names

The compiler requests the name of your Pascal source program. Enter:

Source filename [.PAS]: PRIMES

If you do not enter an extension after the source file name, the compiler attaches the default extension .PAS to the source file name. After you enter a legal file name for the source file, the compiler requests the name of the relocatable object file that it creates during Pass 2. Enter:

Object filename [PRIMES.OBJ]: (press RETURN)

The default name appears in brackets in the above prompt. You can either select this default name by pressing RETURN, or enter the name of the object file you want to create.

The next prompt requests the name of the source listing file. The compiler creates the source listing file during Pass 1 of compilation. The compiler also lists this Pascal source and any compilation errors or warnings as they occur. For this demonstration, send this file to the screen by entering:

Source listing [NUL.LST]: CON

If you press RETURN without entering a source listing file name, the compiler sends the listing file to the null file, NUL, and does not create a source listing. Writing to the NUL file is the same as not writing a file at all. However, error messages always appear on the screen. To specify this default, you need only press RETURN.

Demonstration Run

You can send the source listing to a printer instead of the screen by entering PRN: instead of CON.

The final prompt requests the name of the object listing file. The compiler creates this file during Pass 3 of compilation. For this demonstration, send this file to the screen by entering:

Object listing [NUL.COD]: CON

If you press RETURN without entering an object listing file name, the compiler does not create an object listing file. As a result, you cannot run Pass 3 of the compiler. However, if you enter a file name, the compiler attaches the default extension .COD to the file name.

After you complete your input, Pass 1 of compilation begins. The source listing file appears on the screen as the compiler reads the source file.

During compilation, error messages appear on the screen. For the demonstration program, no error messages should appear. When the compiler finishes, it displays a message similar to the following one:

Errors	Warns	In Pass One
0	0	
Pass One No Errors Detected.		

If your program contains errors in Pass 1, correct the errors immediately and compile the program again. Do not proceed to Pass 2 until pass 1 compiles with no errors.

Pass 1 creates two intermediate files, PASIBF.SYM and PASIBF.BIN. The compiler saves these files on Drive A for use during Pass 2.

STEP 5 -- Run Pass 2 of the compiler

To run Pass 2 of the compiler, you need only enter:

A:B:PAS2

During Pass 2, the compiler performs the following actions:

- Reads the intermediate files, PASIBF.SYM and PASIBF.BIN, that it created in Pass 1.
- Writes the object file.
- Deletes the intermediate files it created in Pass 1.
- If you requested an object listing in Pass 1, the compiler writes two new intermediate files, PASIBF.TMP and PASIBF.OID, to Drive A. The compiler uses these files during Pass 3. If you did not request an object listing, the compiler writes and then deletes the intermediate file PASIBF.TMP during Pass 2. As a result, you would be unable to run Pass 3.

Demonstration Run

When Pass 2 is complete, a message similar to the following one appears on your screen:

```
Code Area Size = #013F ( 319)
Cons Area Size = #001A ( 26)
Data Area Size = #001C ( 28)
Pass Two      No Errors Detected.
```

The first three lines indicate the amount of space that executable code (Code), constants (Cons), and variables (Data) occupy. This figure appears in both hexadecimal and decimal notation. The message "No Errors Detected." refers to Pass 2 only, not the entire compilation.

If your program contains errors in Pass 2, correct the errors immediately and compile the program again. Do not proceed to Pass 3 until Pass 2 compiles with no errors.

Step 6 — Run Pass 3 of the compiler

To run Pass 3 of the compiler, you need only enter:

```
A:B:PAS3
```

During Pass 3, the compiler reads the intermediate files PASIBF.TMP and PASIBF.OID. The compiler then writes the object code listing, CON.COD, to the screen. After the compiler writes this listing, it deletes the two intermediate files. The A: prompt reappears on the screen when Pass 3 is complete.

Step 7 — Link routines in the runtime library to your .OBJ file

After all three phases of compilation are complete, you are ready to link your program. To begin linking, type:

```
A:B:LINK
```

The Linker first requests the name of your relocatable object file. Enter:

```
Object Modules [.OBJ]: PRIMES
```

If you do not attach an extension to the object file name, the Linker assumes the .OBJ extension for the file PRIMES. The next prompts request the names of the run file and the Linker list file. Press RETURN after each prompt to accept the default file specifications in brackets:

```
Run File [PRIMES.EXE]: (press RETURN)
```

```
List File [NUL.MAP]: (press RETURN)
```

Demonstration Run

The final prompt requests the location of the runtime library, PASCAL.LIB. Because this is on Drive B, enter:

Libraries [..LIB]: B:

The Linker then links your program PRIMES.OBJ with the necessary modules in the runtime library. This process creates the executable file PRIMES.EXE on Drive A.

For more information on the Linker and the linking process, refer to The Wang Professional Computer Program Development Guide and Chapter 5 in this manual.

Step 8 -- Run your program

To run your final executable program, enter:

A: PRIMES

The system then executes the program PRIMES.EXE. If the program runs correctly, a listing of all the prime numbers from 1 to 10000 appears on the screen.

Demonstration Run

CHAPTER 4 COMPILING

This chapter supplements the discussion of the compiler in Section 2.2. This chapter contains information about compiler procedures; for a more technical discussion of the compiler, see Section 9.1.

4.1 FILES THE COMPILER WRITES

During compilation, the compiler creates several files. The compiler always creates an object file, which the Linker combines with routines in the runtime module to produce a final executable program. You can also request that the compiler create source listing and object listing files. In addition, the compiler creates and later deletes several intermediate files during compilation. The following paragraphs describe these files.

4.1.1 The Object File

The compiler writes the object file to disk after Pass 2 of the compiler is complete. The object file is a relocatable module; it contains relative, rather than absolute, addresses. The compiler creates this file with the .OBJ extension unless you specify another extension. The Linker links the object module with the Wang PC Pascal runtime library to create an executable module that contains absolute addresses.

4.1.2 The Source Listing File

The source listing file is a line-by-line account of the source file(s), with page headings and messages. A number precedes each line. Any error messages that pertain to that source line refer to this number.

Compiler error messages, shown in the source listing, also appear on your screen. See Appendix H for a list and explanation of all error messages.

If you include files in the compilation with the \$INCLUDE metaccommand, the source listing also contains these files. See Section 26.3 for information on \$INCLUDE; see Section 26.4 for a description of metacommands that control listing file format; and see Section 26.5 for a discussion of features of the listing file.

The various flags, level numbers, error message indicators, and symbol tables make the source listing useful for error checking and debugging. Many programmers prefer to use a printout of the source listing file as a working copy of the program, rather than a printout of the source file itself.

4.1.3 The Object Listing File

The object listing file, a symbolic, assembler-like listing of the object code, lists addresses relative to the start of the program or module. The compiler does not determine absolute addresses until the linking process is complete.

You may want to use the object listing file during program development instead of the source listing file, because:

1. You can look at it to see what code the compiler generates and to familiarize yourself with that code.
2. You can check to see whether assembly language routines would improve program efficiency.
3. You can use it as a guide when you debug your program with the Wang PC Debugger, DEBUG.COM. Refer to The Wang Professional Computer Program Development Guide for an explanation of how to use the Debugger.

4.1.4 The Intermediate Files

Pass 1 creates two intermediate files, PASIBF.SYM and PASIBF.BIN. These incorporate information from your source file and from PASKEY, the Wang PC Pascal predeclarations, for use in creating the object file during Pass 2. The compiler always writes these two intermediate files to the default drive.

Pass 2 reads and then deletes PASIBF.SYM and PASIBF.BIN. Pass 2 then creates one or two new intermediate files, depending on whether you requested an object listing. If, as for the sample session, you plan to run Pass 3 to produce the object listing, Pass 2 writes two intermediate files, PASIBF.TMP and PASIBF.OID.

If you do not request an object listing in Pass 1, Pass 2 writes and later deletes only one new intermediate file, PASIBF.TMP.

Pass 2 assumes that the intermediate files the compiler created in Pass 1 are on the default drive. If you switched disks and they are on another drive, you must indicate their location on the command that starts Pass 2. For example:

```
A: PAS2 A/PAUSE
```

Compiling

The A immediately following the command tells the compiler that PASIBF.BIN and PASIBF.SYM are on Drive A instead of Drive B. The /PAUSE tells the compiler to pause before continuing so that you can insert the disk that contains them into Drive A.

After pausing, Pass 2 prompts:

Press enter key to begin Pass two.

After you insert the new disk in Drive A, press the RETURN key and the compiler proceeds with Pass 2.

The compiler deletes PASIBF.TMP and PASIBF.OID from the default drive during Pass 3. If you change your mind after you request an object listing file and decide not to run Pass 3, be sure to delete these files to recover the space on your disk.

4.2 FILE NAME CONVENTIONS

When you enter the compiler, it prompts you for the names of four files: your source file, the object file, the source listing file, and the object listing file. The only one of these names you must supply is the source file name. This section describes how the compiler constructs the remaining file names from the source file name and how you can override these defaults.

A complete Wang PC file specification has three parts:

1. Drive designation. This specifies the disk drive where the file is or will be. On a single-drive machine, all device names default to Drive A. On multidrive machines, if you do not specify a drive, the compiler assumes the currently logged drive.
2. File name. This is the name you give to a file. Refer to The Wang Professional Computer Introductory Guide for the limitations on assigning file names.
3. File name extension. This system adds this to the file name for further identification of the file. The extension consists of up to three alphanumeric characters preceded by a period. Although you can give any extension to a file name, the Wang PC Pascal Compiler and the Linker recognize and assign certain extensions by default, as shown in Table 4-1.

Table 4-1. Conventional File Name Extensions

Extension	Function of File
.PAS	Wang PC Pascal source file
.FOR	Wang PC FORTRAN source file
.OBJ	Relocatable object file
.LST	Source listing file
.COD	Object listing file
.ASM	Assembly source file
.MAP	Linker map file
.LIB	Library files
.EXE	Run file

If you give unique extensions to your file names, you must include the extension as part of the file name in response to a prompt. If you do not specify an extension, the Wang PC Pascal Compiler supplies one of those shown in Table 4-2.

Table 4-2. File Names Assigned by the Compiler

File	Device	Extension	Full File Name
Source file	dev:	.PAS	dev:filename.PAS
Object file	dev:	.OBJ	dev:filename.OBJ
Source listing	dev:	.LST	dev:NUL.LST
Object listing	dev:	.COD	dev:NUL.COD

Table 4-2 also shows the default file names that the compiler supplies if you give a name for the source file and then press the RETURN key in response to each of the remaining compiler prompts.

The device dev: is the currently logged drive. Even if you specify a device in the source file name, the remaining file specifications default to the currently logged drive. You must specify the name of another drive if that is where you want a particular file to go.

The NUL file is equivalent to creating no file at all; thus, by default, the compiler creates neither a source listing file nor an object listing file. If you enter any part of a file specification in response to either of the last two prompts, the remaining parts default as follows:

Source listing	dev:filename.LST
Object listing	dev:filename.COD

You must explicitly request that the compiler create a listing file. If you specify any nonnull file for the object listing, Pass 2 leaves PASIBF.TMP and PASIBF.OID, the input files for Pass 3, on your work disk until you delete them, either explicitly or by running Pass 3.

If you want to send either listing file to your screen, use one of the special file names USER or CON. Only Wang PC Pascal and Wang PC FORTRAN recognize USER, which writes to the screen immediately as the compiler creates the listing file. All Wang PC programs recognize CON, which saves the screen output and writes it in blocks of 512 bytes.

The rules that govern user input are:

- The system converts all lowercase letters in file names to uppercase letters. For instance, the following names are all identical:

abcde.fgh

AbCdE.FgH

ABCDE.fgh

- To enter a file specification that contains no extension, type the name followed by a period.

Examples:

Source filename [.PAS]: ABC

{ABC.PAS taken as name}

Source filename [.PAS]: ABC.

{ABC taken as name}

- You can enter drive designations and extensions to override the defaults for any parameter. For example, if the currently logged drive is A:, then:

Object filename [ABC.OBJ]: B:

{B:ABC.OBJ is full specification}

- For listing files that default to null, you can accept the default by pressing RETURN, or you can specify a nonnull file by entering any part of a legal file specification. In the latter case, the compiler creates a file with the same default rules that apply to other files. In particular, if you specify a drive or extension, the default base name is the base name of the source file.

Examples:

Source listing [NUL.LST]: (press RETURN)

{NUL: is the default}

Source listing [NUL.LST]: A:

{A:ABC.LST becomes the specification. ABC is part of the source and object filenames.}

- You can enter a semicolon (;) to accept the default values for all remaining parameters. Thus, the quickest way to specify a compilation is:

Source filename [.PAS]: ABC;

You cannot use a semicolon to specify a default source file, since the source file has no default file specification.

- You can use trailing and leading spaces. The following is acceptable:

Source filename [.PAS]: ABC ;

Spaces cannot occur within file names.

4.3. ENTERING THE PASCAL COMPILER

You can enter the Pascal compiler either at command level or from the Program Development menu. To enter the compiler from the Program Development menu, you must first modify this menu with the MODIFY SYSTEM MENUS utility. Refer to The Wang Professional Computer Introductory Guide for a full explanation of the MODIFY SYSTEM MENUS utility.

There are three ways to enter the Pascal compiler at command level:

- Specifying no command line parameters
- Specifying all command line parameters
- Specifying some command line parameters

These methods are discussed below.

4.3.1 Specifying No Command Line Parameters

To enter the compiler without specifying any command line parameters, type:

A:PAS1

Compiling

The compiler then requests values for the four command line parameters as follows:

Source filename [.PAS]: B:MYFILE

Object filename [MYFILE.OBJ]: B:

Source listing [NUL.LST]: MYFILE

Object listing [NUL.COD]: (press RETURN)

In the above example, the file specifications in square brackets indicate the default values the compiler assigns for that parameter. You can accept the default value by pressing RETURN, or override the default by entering a valid file specification. If you enter any part of a file specification here, the compiler creates the file with the same default rules that apply to other files.

If you enter MYFILE in response to the source listing prompt, the name of the listing file becomes A:MYFILE.LST (assuming that the default drive is Drive A).

4.3.2 Specifying All Command Line Parameters

When you enter the Pascal compiler, you can specify all command line parameters in your command string. The general form of such a string is:

A:PAS1 <source>,<object>,<sourcelist>,<objectlist>

The same default naming conventions apply whether you specify command line parameters when you enter the compiler or not. You must separate parameters by commas. If you do not specify a parameter after a comma, the parameter takes on the base name of the source, the default drive designation, and the default extension. Thus, these two command strings are equivalent:

A:PAS1 DATABASE,DATABASE,DATABASE

A:PAS1 DATABASE,,

If you want to accept the normal defaults with null listing files, enter a semicolon (;) after the source file name. Thus, these forms are equivalent:

A:PAS1 YOYO,YOYO,NUL,NUL;

A:PAS1 YOYO;

Spaces and command line switches can occur before or after file names, but not within them.

4.3.3 Specifying Some Command Line Parameters

You can specify some command line parameters when you enter the Pascal compiler. The compiler requests any parameters that you do not specify. For example, you might enter a compiler command line such as:

A:PAS1 TEST,TEST

Source listing [NUL.LST]: (press RETURN)

Object listing [NUL.COD]: (press RETURN)

4.4 PASS 1 COMPILER SWITCHES

By adding switches to the command line when you start Pass 1 of the compiler, or to your response to any of the Pass 1 prompts, you can direct the Wang PC Pascal Compiler to perform additional or alternate functions. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. You can use more than one switch, but each switch must begin with a slash (/). Do not confuse these switches with the Linker switches.

Switches affect the entire compilation. You can place a switch anywhere that you can use spaces. You can enter switches either on the command line or in response to compiler prompts. Table 4-3 shows the compiler switches that are currently available, the default position of the switch, and the corresponding metaccommand.

Table 4-3. Pass 1 Compiler Switches

Switch	Default	Metacommand	Action
/A	off	\$INDEXCK	Checks for array index values in range, including super array indices.
/D	off	\$DEBUG	Switches on all others.
/E	off	\$ENTRY	Generates procedure entry and exit calls for debugger.
/I	off	\$INITCK	Checks for use of uninitialized values.
/L	off	\$LINE	Generates line number calls for debugger.
/M	on	\$MATHCK	Checks for mathematical errors such as overflow and division by zero.
/N	on	\$NILCK	Checks for dereferencing of any pointers that are NIL.
/Q	off	\$DEBUG	Switches off all others.
/R	on	\$RANGECK	Checks for subrange validity including assignments.
/S	on	\$STACKCK	Checks for stack overflow at procedure or function entry.

Because all of the Pass 1 switches correspond to Wang PC Pascal metacommands, you can achieve the same effect either by using the metacommand in the source file or by giving the command as a switch to the compiler. However, any a metacommand you specify in the source file overrides the corresponding switch you give at compile time.

Chapter 26 discusses the Wang PC Pascal metalanguage in detail. The two switches /D and /Q are equivalent to the \$DEBUG+ and \$DEBUG- metacommands, respectively, except that they also turn \$ENTRY and \$LINE on and off.

Several of the switches correspond to runtime error-checking metacommands that end with the letters CK. Turning one of these switches off does not guarantee that the switch does not perform the check; it only means that the switch expends no extra effort to perform the check.

Using the error-checking switches or metacommands significantly increases the amount of code generated. Thus, you may wish to use them in the early stages of program development and later recompile your program without them to reduce code space and decrease execution time.

The following sample command lines and responses illustrate the use of compiler switches (your responses are shown underlined):

To turn on \$INDEXCK:

B: PAS1 /A DEMO,,,NUL

To turn on all of the switches:

B: PAS1 DEMO,,,/D

This sequence first turns all switches off and then turns on \$MATHCK, \$INDEXCK, and \$INITCK:

B: PAS1 DEMO/Q

Object filename [DEMO.OBJ]: /M/A

Source listing [NUL.LST]: DEMO

Object listing [NUL.COD]: /I

CHAPTER 5 LINKING

5.1 FILES THE LINKER READS

A successful Wang PC Pascal compilation produces a relocatable object file. Linking, the next step in program development, is the process of converting one or more relocatable object files into an executable program.

5.1.1 Object Modules

Object files can come from any of the following sources:

- Wang PC Pascal programs, modules, or units
- Wang PC FORTRAN programs, subroutines, or functions
- User code in other high-level languages
- Assembly language routines
- Routines in standard runtime library modules that support facilities such as error handling, heap variable allocation, or input/output

To access Wang PC FORTRAN routines, you must declare the FORTRAN routine EXTERN, and all parameters in the routine must be VARS or CONSTS.

You may need to write assembly language interface routines to translate a Wang PC Pascal or Wang PC FORTRAN calling convention or function return to one assembly language uses. Whatever the language, it must be able to produce linkable object modules. For information on linking assembly languages routines, see Chapter 8. For further information on the Linker, see The Wang Professional Computer Program Development Guide.

The ability to link together programs, units, and modules of Wang PC Pascal source code, as well as assembly language and library routines, means that you can develop a program incrementally. Separate compilation and later linking of separate parts of a program reduces the need for continual recompilation, and also allows you to create programs larger than 64 KB of code (see Chapter 7).

For now, assume that you have created a program that uses one Wang PC Pascal unit and one Wang PC Pascal module and also contains two assembly language external procedures. Assume further that these files were already compiled or, in the case of the assembly language routines, already assembled. The files thus created are:

```

PROG.OBJ
UNIT.OBJ
MODU.OBJ
ASM1.OBJ
ASM2.OBJ

```

To link these all together, enter the Linker by typing:

```
A:LINK
```

The Linker, like the compiler, gives a sequence of four prompts. Before linking can proceed, you must explicitly or implicitly supply the following pieces of information:

- The name(s) of the object modules to be linked
- The name to be given to the executable run file
- The Linker listing file
- The names of any libraries to be searched (other than PASCAL.LIB)

As with the compiler, responses to all except the first prompt can be defaults.

In response to the first Linker prompt, enter the names of the object files, separated by plus signs, as shown:

```
PROG+UNIT+MODU+ASM1+ASM2
```

The first object file in the above response must be a Wang PC Pascal program, module, or unit, although it need not be the main program. Do not put any assembly language module first; the Linker may order segments incorrectly. After the initial Wang PC Pascal object file, you can list the other modules, units, or assembly language routines in any order.

Type a semicolon after the name of the last object file you wish to link to tell the Linker to omit the remaining prompts and to supply defaults for all remaining parameters. Table 5-1 lists the defaults the Linker supplies.

Table 5-1. Linker Defaults

Prompt	Default Response
Object modules	None
Run file	prog.EXE
List map	NUL.MAP
Libraries	PASCAL.LIB

5.1.2 Libraries

A runtime library contains runtime modules that must be present during linking to resolve references made during compilation. The Wang PC Pascal Compiler generates space for instructions for most floating-point operations. During linking, these instructions are resolved using information in the runtime library.

Because Wang PC Pascal is designed for use on machines with or without an 8087 coprocessor, the compiler provides two versions of the runtime library:

- If you use the library PASCAL.L87 to link the program, the space assigned by the compiler becomes instructions for the 8087 coprocessor. The program will only run correctly with an 8087 coprocessor.
- If you use the library PASCAL.LIB to link your program, the compiler transforms program instructions into emulator interrupts. The Linker generates additional program code that services these interrupts.

NOTE:

The current implementation of the Wang PC does not support the 8087 numeric coprocessor. The 8087 coprocessor will, however, be available for use with future versions of the Wang PC.

You can specify that the Linker search libraries in addition to PASCAL.L87 and PASCAL.LIB; see The Wang Professional Computer Program Development Guide for information.

If you press the RETURN key in response to the final Linker prompt, the Linker automatically searches for a library called PASCAL.LIB on the default drive. If PASCAL.LIB is not on the default drive, the following message appears on your screen:

```
Cannot find library PASCAL.LIB
Enter new drive letter:
```

Switch disks if necessary, and then type the name of the drive that does contain PASCAL.LIB. If instead you respond by pressing the RETURN key, linking proceeds without a library search. You can also achieve this by using the Linker option switch -NO (for -NODEFAULTLIBRARYSEARCH), to override the automatic search for PASCAL.LIB. This produces unresolved reference error messages unless you replace every required runtime routine with a routine of your own.

To instruct the Linker to search other libraries (for example, FORTRAN.LIB) as well as PASCAL.LIB, give the library names, separated by plus signs, in response to this prompt. See The Wang Professional Computer Program Development Guide for complete information on using different libraries with the Linker.

5.2 FILES THE LINKER WRITES

The primary output of the linking process is an executable run file. You can also request a Linker map or listing file, which serves much the same purpose as the compiler listing files. The Linker also writes and later deletes one temporary file.

5.2.1 The Run File

The run file the Linker produces is your executable program. The default file name, given in brackets as part of the prompt, is the name of the first module you list in response to the first prompt. To accept this prompt, press RETURN. To specify another run file name, type in the name you want. All run files receive the extension .EXE, even if you specify something else.

The Linker ordinarily saves the run file, with the extension .EXE, on the disk in the default drive. To specify another drive, type a drive name in response to the run file prompt. This may be necessary if your program is large.

5.2.2 The Linker Listing File

The Linker map (also called the Linker listing file) shows the addresses relative to the start of the run module for every code or data segment in your program. If you request it with the -MAP switch, the Linker map can also include all EXTERN and PUBLIC variables. Refer to The Wang Professional Computer Program Development Guide for an explanation of the -MAP switch.

The Linker map defaults to the NUL file. You can also obtain a printout of this map, have the Linker display it on the screen, or save it on diskette. In the early stages of program development, you may find it useful to inspect the Linker map in these two instances:

- When you use the Debugger to set breakpoints and locate routines and variables
- To find out why a load module is large (for example, what routines are in the module, how big they are, and what they contain)

As the prompt indicates, the default for the Linker map is the NUL file (that is, no file at all). Press RETURN to accept this default. If you wish to see the Linker map but do not want to write it to a disk file, type CON in response to the list file prompt. (The Linker does not recognize the special file name USER.) If you want to write the file to disk, give a device or file name.

5.2.3 VM.TMP

Linking begins after you respond to all of the Linker prompts. If the Linker needs more memory space to link your program than is available, it creates a file called VM.TMP on the disk in the default drive and displays a message:

```
VM.TMP has been created.
Do not change disk in drive B:.
```

If the Linker uses up the additional space or if you remove the diskette that contains VM.TMP before linking is complete, the Linker aborts processing.

When the Linker finishes, it erases VM.TMP from the disk and displays any errors that occurred during linking. For a list of the Linker error messages, refer to The Wang Professional Computer Program Development Guide.

If the Linker aborts processing, use the DOS Command Processor utility DIR to check the contents of your diskette to make sure that the Linker deleted VM.TMP. Then, to make sure the space is free, use the DOS Command Processor utility CHKDSK. CHKDSK reclaims any available space from unclosed files and tells you the total amount of available space on the diskette. Refer to The Wang Professional Computer Introductory Guide for an explanation of the DIR and CHKDSK utilities.

5.3 LINKER SWITCHES

After any of the Linker prompts, you can give one or more Linker switches. See The Wang Professional Computer Program Development Guide for more information on Linker switches and when and how to use them.

The -PAUSE switch is particularly useful for linking large Wang PC Pascal programs, because it allows you to switch diskettes before writing the run file. If the Linker creates a VM.TMP file, however, you must not switch the diskette in the default drive.

NOTE:

For Wang PC Pascal and Wang PC FORTRAN programs, do not use the Linker switches -HIGH or -STACK.

CHAPTER 6

USING A BATCH COMMAND FILE

The DOS Command Processor allows you to create a batch file that executes a series of commands. Creating and using batch command files is described fully in The Wang Professional Computer Introductory Guide. This chapter provides a brief description of command files in the context of compiling, linking, and running a Wang PC Pascal program.

A batch command file is a text file of lines that are DOS commands. If a batch file is already open when the DOS Command Processor is ready to process a command, the next line in the batch file becomes the command line. After processing all batch command lines (or if you terminate processing by pressing CONTROL + C), the DOS Command Processor goes back to reading command lines from the screen.

The compiler, the Linker, or a user program cannot read batch file lines. Thus, you cannot put responses to file name prompts, \$INCONST values, or the like in a batch file. You must enter all compiler parameters on the command line, as described in Section 4.3.2.

The batch file can contain dummy parameters. When you execute the batch file, the compiler replaces the dummy parameters with actual parameters. The symbol %1 refers to the first parameter on the command line, %2 to the second parameter, and so on. The limit is %9. A batch command file must have the extension .BAT and should be on either the program disk or the utility disk.

A PAUSE command consists of the word PAUSE, followed by the text of the prompt that you want to appear on the screen. This command tells the operating system to pause, display the prompt that you have defined, and wait for some further input before continuing. This input is the actual parameters that replace the dummy parameters in the batch file.

If your program is already debugged and you are making only minor changes to it, you can speed up the compilation process by creating a batch file that issues the compile, link, and run commands. For example, use the Wang PC Editor (refer to The Wang Professional Computer Program Development Guide) to create the following batch file, COLIGO.BAT:

```
A:PA$1 %1,,,;
PAUSE ...If no errors, insert PAS2 disk in Drive A.
A:PA$2
PAUSE ...Insert runtime libraries disk in Drive A.
A:LINK %1;
%1
```

Using a Batch Command File

To execute this file, type:

COLIGO SORT

SORT is the name of the source program you want to compile, link, and run. The commands in the batch file then instruct the system to perform the following actions.

1. The first line runs Pass 1 of the compiler.
2. The second line generates a pause and prompts you to insert the Pass 2 disk.
3. The third line runs Pass 2.
4. The fourth line generates a pause and prompts you to insert the runtime library.
5. The fifth line links the object file.
6. The sixth line runs the executable file.

A .BAT file executes only if no .COM file or .EXE file with the same name exists. If you keep your source file and .BAT file on the same disk, they must have different file names.

For more information about batch command files, see The Wang Professional Computer Introductory Guide.

CHAPTER 7

COMPILING AND LINKING LARGE PROGRAMS

Occasionally, you may find that a large program exceeds one or more of the program size limitations imposed by the compiler, the Linker, or your machine. This chapter describes some ways to avoid or work within such limits.

7.1 LIMITS ON CODE SIZE

The upper limit on the size of code that the Wang PC Pascal Compiler can generate at one time is 64 KB. Since you can compile any number of compilands separately and link them together later, however, the real program size limit is not 64 KB but the amount of memory available. For example, you can separately compile six different compilands of 50 KB each. Linking them together produces a program with a total of 300 KB of code.

In practice, a source file large enough to generate 64 KB of code would be thousands of lines long, and unwieldy both to edit and to maintain. A better practice is to break a large program into Wang PC Pascal modules and units to better structure the development and maintenance process. As always, there is a tradeoff between size and speed. Procedure and function calls within a module to routines without the PUBLIC attribute are somewhat faster than calls to routines that contain the PUBLIC attribute. This happens because the PUBLIC attribute contains intersegment calls, which run slower than intrasegment calls found in routines without the PUBLIC attribute.

7.2 LIMITS ON DATA SIZE

Data includes your main variables, the stack, and the heap. Wang PC Pascal operates with data in two regions of memory: the default data segment, and the segmented data space. The upper limit on the amount of data that can reside in the default data segment is also 64 KB; an array in this segment can occupy no more than 32 KB. You can go beyond this limit by placing certain kinds of data outside the default data segment, using ADS variables, VARS and CONSTS parameters, and segmented ORIGIN variables.

The default data segment normally holds:

- All statically allocated variables
- Constants that reside in memory

- Heap variables
- The stack, which holds parameters, return addresses, stack variables, etc.

Although operations with data in the default data segment are more efficient (that is, they generate less code and run faster) than those with data that may be in any segment, almost all Wang PC Pascal operations work equally well on data outside the default data segment.

The segmented data space includes the entire 8086 address space, including the default data segment. ADS (segmented address) variables, VARS and CONSTS parameters, and segmented ORIGIN variables can reference data outside the default data segment.

Data must reside in the default data segment only in the following cases:

- File variables
- The LSTRING parameters to ENCODE and DECODE
- All parameters to READSET

To allocate data outside the default data segment, you must go outside the Wang PC Pascal system itself. If you already know the address of free blocks of memory on your computer, you can use these addresses in a segmented ORIGIN attribute or assign them to an ADS variable. Otherwise, you can get the addresses of free memory by moving a pair of ADS variables to the lower and upper bounds of available memory. (Refer to Section A.1.)

Many applications use a large block of memory for primary data, and variables residing elsewhere in memory to process this data. For example, a text editor has a work area, a data base system has a data area (or index area), and so on. ADS variables can manage this large block outside the default data segment.

In the default data segment, the space that the heap and the stack occupy in memory increase in size and grow toward each other. Heap allocation attempts to use existing disposed blocks in the heap itself before growing into memory shared with the stack.

As a part of this process, heap allocation merges adjacent disposed blocks, and free blocks at the end of the heap become available to the stack. However, only heap allocation (such as NEW or GETHQQ) releases free heap blocks to the stack. Therefore, if you are running out of stack after a number of DISPOSE operations, make the following call:

```
EVAL (GETHQQ (65534));
```


7.3 WORKING WITH LIMITS ON COMPILER MEMORY

During compilation, large programs can reach the compiler memory limits because a source file contains many identifiers, or because of the complexity of the program itself. If a program reaches one of these limits, the error message "Compiler Out Of Memory" appears.

There is no limit on number of bytes in a source file. The maximum number of lines is 32767; however, any source file this big runs into other limits before it reaches the line limit.

7.3.1 Identifiers

Pass 1 of the compiler can manage a maximum of about a thousand identifiers visible at any one time. This assumes a 64 KB default data segment (about 160 KB of memory total); it also assumes that most of your identifiers are seven characters or shorter and are not PUBLIC or EXTERN.

After compilation, the compiler can reduce local identifiers in a procedure or function to provide room for new ones. Several methods of reducing the number of identifiers in a program are:

1. Break your program into modules or units

The best way to reduce the number of identifiers is to break up your program into modules or units. When dividing your application into pieces, one guiding principle is to minimize the number of shared (PUBLIC and EXTERN) identifiers. A program with a small number of shared identifiers compiles faster than a program of the same size containing a large number of shared identifiers.

Breaking up a program may force you to choose between a shared variable and a shared procedure or function. Usually, a shared procedure or function is "cleaner" than a shared variable; it is easier to trace the use of a procedure than the use of a variable, for example. However, a shared variable is usually more efficient in terms of memory required and number of identifiers used.

2. Simplify your identifiers

Although it reduces the readability of a program (because naming something is a more readable way of referring to it than giving an arbitrary number), you can simplify your identifiers by replacing names with numbers. If necessary, any of the following may help:

- a. Change enumerated types into WORD types and use numbers instead of identifiers.
- b. Use constant literals instead of constant identifiers.
- c. Combine related procedures and functions into single ones, with a parameter indicating the type of call.

Compiling and Linking Large Programs

- d. Combine variables into an array and refer to the variables using constant array indices.

3. Remove unneeded identifiers from PASKEY

You can also remove identifiers of predeclared procedures and functions you do not need from PASKEY, at least those procedures and functions in the final section. You must, however, remove an identifier in this section from the UNIT list, from the interface (the declaration itself), and from the USES list. However, you cannot remove the identifier FCBFQQ.

You can also remove identifiers of intrinsic procedures and functions. These are a list near the start of PASKEY from READLN to RESULT. You must replace any identifiers you remove with an asterisk (*). You must not remove the procedure READFN if you have program parameters.

Finally, you can remove the following declarations:

ADAPQQ	INTEGER2
ADRMEM	MAXINT
ADSMEM	MAXINT4
BYTE	MAXWORD
INTEGER1	BYTE

Removing any other identifiers from PASKEY generates the error message "144 Compiler Internal Error".

When an interface USES another interface, it must import all identifiers in the other interface. To do this, the other interface must declare all identifiers; thus, its identifiers occur twice. If a third interface USES both of the first two, the first interface's identifiers occur three times and the second interface's identifiers occur twice, and so on. This is an easy way to run out of identifiers!

The only reason an interface needs to USE another interface is to import identifiers for types; an interface has no use for variables, procedures, and functions. You can declare a single interface with global types; this is the only interface that other interfaces use. Once compilation passes the USES clause in the PROGRAM, MODULE, or IMPLEMENTATION, the compiler removes many of these "extra" identifiers.

7.3.2 Complex Expressions

You can also run out of memory in Pass 1 with any of the following:

- A very complex statement or expression (that is, one that contains many levels of nesting)

- A large number of error messages
- A large number of structured constants, including string constants

You may be able to change literal strings and other structured constants into EXTERN READONLY variables which another module initializes as PUBLIC variables.

If a program completes Pass 1 without running out of memory, it can usually complete Pass 2. The major exception to this rule occurs with complex basic blocks, as in either of the following:

- Sequences of statements with no labels or other breaks
- Sequences of statements containing very long expressions or parameter lists (especially a WRITE or WRITELN procedure call with many expressions)

If Pass two runs out of memory, the error message: "Compiler Out Of Memory" appears, along with a message stating at which program line the compiler ran out of memory. If there is a particularly long expression or parameter list near this line, break it up by assigning parts of the expression to local variables (or using multiple WRITE calls). If the compiler still runs out of memory at this line, add labels to statements to break the expression up even further.

7.4 WORKING WITH LIMITS ON DISK MEMORY

You may also encounter limits in the number of disk drives on your computer or the maximum file size on one disk. As with other limits, there are several possible ways to circumvent these limits. The simplest method to avoid these limits is to load a compiler pass, and then switch disks and run the pass.

7.4.1 Pass 1

For PAS1.EXE, type PAS1 (or dev:PAS1, if necessary) to load Pass 1 and read the PASKEY file. When the Source File prompt appears, you can remove the disk that contains PAS1.EXE and PASKEY. If you have a single-drive system, replace the system disk with the disk that contains your source file. PAS1 writes its intermediate files on the same disk.

If you have a dual-drive system, insert your source file in the drive that is not the default drive. Because the compiler writes the intermediate files to the default drive, you must give an explicit drive letter for your source file. A source listing file typically goes on the same drive as the source.

If your source file does not fit on one disk, you can break it into pieces and use the \$INCLUDE metaccommand to compile the pieces as a group. One way to do this is to create a master file with lines such as:

```
{MESSAGE:'Insert B:P1.PAS'
  $INCONST:P1 $INCLUDE:'B:P1.PAS'}
{MESSAGE:'Insert B:P2.PAS'
  $INCONST:P2 $INCLUDE:'B:P2.PAS'}
{MESSAGE:'Insert B:P3.PAS'
  $INCONST:P3 $INCLUDE:'B:P3.PAS'}
```

The \$INCONST metaccommand makes the compiler pause while you switch disks. You can also enter these \$INCLUDE metacommands. Give USER as the name of your source file and type your \$INCLUDE metacommands directly, one on each line. To end the compilation, type CONTROL + Z (end-of-file).

If your source file does not fit on one disk, neither will your source listing file. You must send it directly to the printer. If a listing file will fit on the disk, except that the source and intermediate (PASIBF) files take up too much room, include a line such as the following near the start of your source file:

```
{INCONST: ZERROR} CONST ERROR = 1 DIV ZERROR;
```

If you respond with 0 to the Inconst ZERROR prompt, a compiler error occurs. The compiler error stops the writing of the intermediate files; this leaves room on the disk for your listing. However, you must then run the front end twice, once to generate intermediate files for Pass 2 and once for the listing.

Another way to control a large listing file is the \$LIST metaccommand. Stop generating listing code with the \$LIST- metaccommand, and then use the \$LIST+ and \$LIST- metacommands to bracket those portions of the program for which you want a source listing.

7.4.2 Pass 2

Two command line parameters available with Pass 2 can help you with disk limitations:

1. You can indicate a drive letter on which your input intermediate files, PASIBF.SYM and PASIBF.BIN, reside.
2. The /PAUSE switch tells Pass 2 to pause while you remove the disk that contains PAS2.EXE and insert another disk.

For example, if you have a single-drive system, insert your PAS2.EXE disk and type PAS2 /PAUSE. After you load PAS2.EXE, the message "Press ENTER key to begin Pass two" appears. Remove the PAS2.EXE disk and insert the disk with the intermediate file from Pass 1. Press the RETURN key, and Pass 2 runs.

If you have two drives, but you run out of disk memory when executing Pass 2, you need to have the input intermediate files PASIBF.SYM and PASIBF.BIN on one drive and PASIBF.TMP on the other drive (also PASIBF.OID if you are making an object listing file). The compiler writes the PASIBF.TMP file and the PASIBF.OID file used in Pass 3 to the default drive.

Give Pass 2 a drive letter to specify the drive that contains the PASIBF.SYM and PASIBF.BIN files; for example, "PAS2 B". Normally, you also need the pause command; for example, "PAS2 B/PAUSE". Pass 2 responds with a message such as the following: "PASIBF.SYM and PASIBF.BIN are on B:". The pause prompt follows this message: "Press enter key to begin Pass two". When you run Pass 2 with the PASIBF files on two disks, the object file usually goes on the same disk as PASIBF.TMP and PASIBF.OID (that is, in the default drive). If it does not fit, and you are making an object listing file, you can compile your program twice, once without the object listing but with the object file itself, and once with an object listing but with NUL used for the object file.

7.4.3 Linking

If your program occupies more than one diskette, or if your Wang PC only has one diskette drive, you may run into similar problems when you link your program. Because you can split your program into pieces and compile them separately (although you must link the entire program at one time) you may run into disk limitations in the Linker but not the compiler.

The Linker prompts you for any object files and/or libraries it cannot find, so you can insert the correct disk and continue linking. Also, the -PAUSE switch makes the Linker wait after linking but before writing the run (.EXE) file, so you can create a run file that fills an entire disk. However, creation of the virtual file VM.TMP and the link map limit the amount of disk-changing you can do.

On a single-drive system:

1. Load the Linker by typing LINK.
2. Remove the disk that contains LINK.EXE and insert the disk that contains your object file(s) and, if there is room, any libraries.
3. Respond to the Linker prompts, but include the -PAUSE switch with the run file if you want the run file on another disk.

Unless all object files, libraries, and the run file fit on one disk, you must not write the Linker listing to a disk file. Instead, send the Linker map to NUL, CON, or directly to your printer. Because the Linker writes its map at various points in the linking process, you cannot change disks while the Linker is writing a Linker map.

The Linker prompts you when it needs an object file, a library file, or is about to write the run file; exchange disks as necessary when this happens. If the Linker gives a message that it is creating VM.TMP, its virtual memory file, you can no longer switch disks; thus, you may not be able to link without more memory or a second disk drive.

With two disk drives, you can devote one drive (the default) to the VM.TMP file (and to the Linker map, if you want one). Use the other drive for your object files, libraries, and run file (using the -PAUSE switch). With this method, you can link very large programs.

The Linker makes two passes through the object files and libraries: one to build a symbol table and allocate memory, and one to actually build the run file. This means you insert a disk that contains object files or libraries twice, before you insert the disk that will receive your run file.

7.4.4 A Complex Example

The following example illustrates compiling and linking a very large program. The example assumes that the machine has two drives and that you do not want any of the listing files.

Pass 1

1. Log onto Drive B and insert a disk containing only PASKEY in Drive B.
2. Insert the disk that contains PAS1.EXE in Drive A. Type A:PAS1, and wait for the Source File prompt.
3. Remove the disk that contains PAS1.EXE from Drive A. Insert the disk that contains the source file LARGE.PAS.
4. Respond to the Source File prompt with A:LARGE,A:LARGE, and wait for Pass 2 to run.

Pass 2

1. Log onto Drive A. Remove the source disk from Drive A.
2. Insert the disk that contains PAS2.EXE in Drive A. Type PAS2 B/PAUSE and wait for the Pass 2 prompt.
3. Remove the disk that contains PAS2.EXE from Drive A. Insert an empty disk to which the compiler will write the object file.
4. Respond to the Pass 2 prompt by pressing the RETURN key and wait for Pass 2 to run.
5. Remove the disk that contains the object file from Drive A.

Linking

1. Log onto Drive B, which now contains an empty disk.
2. Insert LINK.EXE in Drive A; type A:LINK and wait for the Object Modules prompt.

3. Remove the disk that contains LINK.EXE from Drive A and insert the disk that contains the object file(s).
4. Respond to the Object Modules prompt by typing A:LARGE (plus any other object files).
5. Respond to the Run File prompt by typing LARGE/PAUSE.
6. Respond to the List File prompt by pressing the RETURN key, or type B:LARGE to get a Linker map.
7. Respond to the Libraries prompt by pressing RETURN or with a library name (the library must be on Drive A).
8. Wait for Linker to run, changing the Drive A disk after prompts as necessary.

7.5 MINIMIZING LOAD MODULE SIZE

You can reduce the size of some Wang PC Pascal load modules by eliminating runtime modules your program does not use. You can make reductions in several areas:

- Input/output (I/O)
- Runtime error messages
- Real number operations
- Debugging

7.5.1 I/O

Because most Wang PC Pascal programs perform I/O, they require linking to the Wang PC Pascal file system in the runtime library. Some programs do not perform I/O, however, and others perform I/O by directly calling the Wang PC Pascal Unit U file routines or calling operating system I/O routines. (For more information on Unit U, see Section 9.2.) Nonetheless, all programs include calls to INIFQQ and ENDYQQ, the procedures that initialize and terminate the file system. These calls increase the size of the load module by linking and loading routines that the program may never use.

If a program does not need the file system routines, you can eliminate unnecessary file support by declaring dummy INIFQQ and ENDYQQ subroutines in your program, as follows:

```

PROCEDURE INIFQQ [PUBLIC];
BEGIN
END;

PROCEDURE ENDYQQ [PUBLIC];
BEGIN
END;

```

The Linker still loads the Unit U procedures necessary to access the terminal (INIYQQ, ENDUQQ, PTYUQQ, PLYUQQ, and GTYUQQ), so that the runtime system can write any runtime error messages.

If you do include the dummy procedures shown, and the Linker produces any error messages for global names that end with the FQQ or UQQ suffix, your program does not require the file system and the process described above does not work.

If your program does not require the I/O-handling procedures that Unit U calls, you can use the dummy file NULF.OBJ instead. NULF.OBJ contains the dummy subroutines for INIFQQ and ENDYQQ, as well as dummies for INIYQQ and ENDUQQ.

7.5.2 Runtime Error Handling

If your program does not need runtime error handling, you can reduce the size of the load module by eliminating the error message module and replacing it with the null object module, NULE6.OBJ. NULE6.OBJ provides for simple termination of a program if an error occurs.

INUXQQ, the unit initialization helper, also resides in the error unit. To replace error handling with NULE6, you must initialize any unit yourself and remove the keyword BEGIN from all the interfaces in your source program.

7.5.3 Real Number Operations

If a Wang PC Pascal program does no real number operations, it does not require INIX87 and ENDX87, the modules that initialize and terminate the real number support system. The dummy object module NULR7.OBJ provides dummy routines for these two modules.

7.5.4 Error Checking

Compiling and linking a program with the error-checking switches or metacommands on may generate up to 40 percent more code (even more with \$LINE+) than with these switches or metacommands off. Therefore, you successfully compile, link, and run a program, turn the error-checking switches off and do the entire process again to create a program that runs considerably faster.

Compiling and Linking Large Programs

CHAPTER 8 USING ASSEMBLY LANGUAGE ROUTINES

After describing the Wang PC Pascal calling conventions and internal representations of data types, this chapter shows how to interface 8086 assembly language routines to Wang PC Pascal compilands. The information in this chapter is not required for most Wang PC Pascal programs and is intended primarily for the advanced programmer who is familiar with the following material:

- The `EXTERN` directive. Refer to Chapter 22.
- Procedure and function parameters. Refer to Chapter 22.
- Wang PC Assembly. Refer to The Wang Professional Computer Assembly Language Reference Manual (700-8315).

8.1 CALLING CONVENTIONS

At runtime, each active procedure or function has a "frame" allocated on the stack. The frame contains the data shown in Figure 8-1.

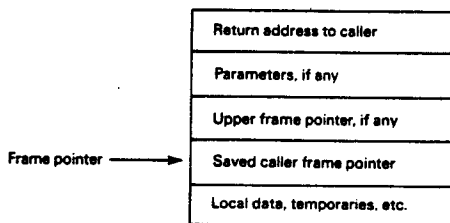


Figure 8-1. Contents of the Frame

The frame pointer points at the saved caller frame pointer, below the return address, and is used to access frame data. A procedure or function nested within another procedure or function has an upper frame pointer, so it can access variables in the statically enclosing frame.

The following takes place during a procedure or function call:

1. The caller saves any registers it needs (except the frame pointer).
2. The caller pushes parameters in the same order as they are declared in the source and then performs the call.
3. The called routine pushes the old frame pointer, sets up its new frame pointer, and allocates any other stack locations needed. It also checks for adequate stack space if \$STACKCK was on.

To return to the calling routine, the called routine restores the caller's frame pointer, releases the entire frame (including parameters), and then returns to the calling routine. Not all of these steps need necessarily be taken in an assembly language routine. You must only ensure that the frame pointer is not modified and that the entire frame, including all parameters, is popped off the stack before returning. For information on the assembly language interface, refer to Section 8.3.

The standard entry and exit sequences (with \$STACKCK-) are as follows:

```
PUSH    BP
MOV     BP,SP
<body of routine>
MOV     SP,BP
POP     BP
RET     PARAMETERSIZE
```

A function always returns its value in registers. For real types, structured types, and pointers to super arrays, regardless of length, the caller allocates a temporary frame for the result and passes the offset address to the function like a parameter. When the called routine returns, it places the address back in the normal return register (AX).

8086 and 8088 microprocessors perform a long call if the called routine is PUBLIC or EXTERN. In all other cases, they perform a short call.

The called routine must save the BP register, which contains the Wang PC Pascal frame pointer, as well as save the DS segment register. Interrupt routines, both user-declared and 8087 support, use the SS register to locate the default data segment. Thus, the called routine must not change the SS register if interrupts are enabled, and need not save other registers (FLAGS, AX, BX, CX, DX, SI, DI, and ES).

Functions return a 1-byte value in AL, a 2-byte value in AX, and a 4-byte value in DX:AX (high part:low part, or segment:offset).

8.2 INTERNAL REPRESENTATIONS OF DATA TYPES

This list describes the internal representation of Wang PC Pascal data types.

1. INTEGER and WORD. INTEGER values are 16-bit two's complement numbers. INTEGER data types allocate an 8-bit byte to a subrange that requires 8 bits or less (in the range -127 to 127). WORD values are 16-bit unsigned numbers. A WORD data type allocates an 8-bit byte to a subrange in the range 0..255. For 16-bit INTEGER and WORD values, the least significant byte has the lower, even address.
2. INTEGER4 and REAL. INTEGER4 values are 32-bit two's complement numbers, with the least significant byte at the lowest, even address and more significant bytes at increasing addresses. There are no subranges for INTEGER4.

IEEE 4-byte real numbers have a sign bit, 8-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives an exponent range of 10^{+38} and 7 digits of precision. The maximum real number is normally 1.701411E38.

IEEE 8-byte real numbers have a similar format, except that the exponent is 11-bits excess 1023, and the mantissa has 52 bits (plus the implied high-order 1 bit). This gives an exponent range of 10^{+306} and 15 digits of precision.

In either case, a number with an exponent of all zeros equals zero. An exponent of all ones is a flag for an invalid real number, or "not a number" (NaN).

3. CHAR, BOOLEAN, and enumerated types. CHAR values and BOOLEAN values take 8 bits. CHAR values correspond to the ASCII collating sequence. For BOOLEAN values, FALSE is 0 and TRUE is 1. The low-order bit (bit 0) generally checks this value. Bits 1 through 7 are FALSE.

Enumerated types less than or equal to 256 values take 8 bits; otherwise, they take 16 bits. Values start at 0. Subrange values take either 8 or 16 bits.

4. Reference types. Pointer values currently take 16 bits. A pointer is a default data segment offset. A pointer to a super array type precedes the bounds (see Item 6 of this list), increasing the length of the pointer value (DS/SS).

ADR and ADS are offset addresses and segmented addresses, respectively. For segmented addresses, the offset is the lower address, and the segment follows.

The heap contains heap blocks, which can be allocated or free. A heap block contains a header WORD, with a 15-bit length (in WORDS) and the lower-order bit, which is on for free blocks and off for allocated blocks. The starting and ending heap addresses are WORD variables in BEGHQQ and ENDHQQ.

5. Procedural and functional parameters. Procedural parameters contain a reference to the procedure or function's location along with a reference to the "upper frame pointer" (a list of stack frames of statically enclosing routines). The parameter always contains two words, in one of two formats. In the first format, the first word contains the actual routine's address (a local code segment offset), and the second word contains the upper frame pointer. The upper frame pointer is zero if the actual routine does not reside within a procedure or function and, therefore, the routine has no upper frame pointer.

In the second format, used for segmented address targets, the first word is zero and the second word contains a data segment offset address. This is an offset to two words in the constant area that contain the segmented address of the actual routine. There is never an upper frame pointer in this case.

6. Super arrays. A super array representation is similar whether it is a reference parameter or the referent of a pointer. The address (reference parameter) or pointer value, which is either 2 or 4 bytes long, comes first. The upper bounds, which are signed or unsigned 16-bit quantities, follow the address. The bounds occur in the same order as they are declared. A pointer value to a super array type is normally longer than other pointers, since the upper bounds are included.

7. Sets. The number of bytes allocated for a SET is:

$$(\text{ORD (upperbound)} \text{ DIV } 16) * 2 + 2$$

This is always an even number from 2 to 32 bytes. For example, SET OF 'A'..'Z' requires 12 bytes. Internally, a set consists of an array of bits, with one bit for every possible ORD value from 0 to the upper bound. Bits in a byte are accessed starting with the most significant bit. The occurrence of a given ORD value as an element of a set implies the bit is 1, and the byte and bit position of a given ORD value of any set is the same. For example, the ORD value of 'A' is 65, and the second bit (2#01000000) of the ninth byte in a set is 1 if 'A' is in the set.

8. Files. A FILE data type in a program is a record called a file control block (of type FCBFQQ) in the file unit. The initial portion of the FCBFQQ record is standard for all files, but the remainder is available for use by the particular target file system. The end of the file control block contains the current buffer variable. The internal form of a file varies depending on the target file system.

Under the DOS Command Processor, ASCII files consist of lines followed by a carriage return and linefeed pair, which together are a "line marker." Binary files are simply a stream of bytes.

9. Structures. The internal form of arrays and records consists of the internal forms of the components, in the same order as in the declaration. Arrays, records, variants, sets, and files always start on a word boundary. In any case, variables cannot occupy more than MAXWORD (64 KB). A PACKED data type has the same representation as an unpacked data type.

A variable or component that is 16 bits or larger is always aligned on a word boundary; therefore, it always has an even byte address. The only exception is when you specify explicit field offsets in a program.

An 8-bit variable is also aligned on a word boundary, but an 8-bit component of a structure (array or record) is aligned on a byte boundary, which can be at an even or odd address.

Some variables are initialized automatically, whether they reside in fixed memory, on the stack, or on the heap.

1. Files (FCBFQQ records) are initialized by calling NEWFQQ, by passing the size of a text file line buffer or binary file component, and by passing a Boolean flag value to indicate whether the file is a text file.
2. If \$INITCK is on, it initializes INTEGER values and their 2-byte subranges to 16#8000, 1-byte INTEGER subranges to 16#80, IEEE REAL values to 16#FFFF, and pointers to 16#0001. \$INITCK never initializes the following variables.
 - a. Variables found in a VALUE section
 - b. Variant fields in a record
 - c. Super arrays allocated on the heap

The compiler generates the extra code necessary to initialize stack and heap variables.

8.3 INTERFACING TO ASSEMBLY LANGUAGE ROUTINES

In general, you declare interfaced procedures and functions `EXTERN` in the Wang PC Pascal source. When you call an `EXTERN` procedure or function, actual parameters are pushed on the stack in the order in which they are declared. If a parameter is a value parameter, an actual value is pushed on the stack.

If a parameter is a `VAR` or `CONST` reference parameter, the address of the variable is pushed on the stack. Only the 2-byte offset is pushed, and not the segment. The offset is within the default data segment, `DS` (where `SS = DS`). In contrast, a `VARS` or `CONSTS` parameter includes both a 2-byte segment and a 2-byte offset, with the segment pushed first.

Super array reference parameters include their upper bounds, pushed as value parameters before the address is pushed. For multidimensional super arrays, bounds are pushed in reverse order (the last flexible bound is pushed first).

For some functions, a final, hidden offset address for the return value temporary variable is pushed last.

After all parameters are pushed, a far call instruction pushes the return address for `PUBLIC` and `EXTERN` procedures. The return address is segmented, so the segment is pushed first, followed by the offset. This is the general starting state of the stack for any assembly language routine that wishes to access parameters.

For example, assume that you have created and compiled the following program, which contains the `EXTERN` function `ADD`:

```
PROGRAM ASM_INTERFACE (INPUT, OUTPUT);
VAR I, TOTAL : INTEGER;
FUNCTION ADD (VAR A:INTEGER; B:INTEGER):
    INTEGER; EXTERN;
BEGIN
    I :=10;
    TOTAL := ADD (I, 15);
    WRITELN (OUTPUT, TOTAL)
END.
```

When the program executes the `ADD` function at runtime, it sets up the stack as shown in Figure 8-2.

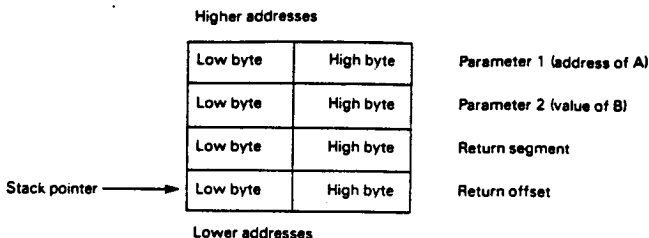


Figure 8-2. Stack Before Transfer to ADD

Before you can run such a program, however, you must link it to a routine that implements the ADD function. Implementation of ADD in assembly language might look like this:

```

DATA    SEGMENT PUBLIC 'DATA'
        ;PUBLIC and EXTERN data declarations go here.
DATA    ENDS
DGROUP  GROUP DATA
        ASSUME  CS:ADDS,DS:DGROUP,SS:DGROUP

ADDS    SEGMENT 'CODE'
PUBLIC  ADD
ADD     PROC FAR
        PUSH BP           ;Save frame pointer on stack
        MOV  BP,SP        ;Address parameters
        MOV  AX,6[BP]     ;AX := value of B
        MOV  BX,8[BP]     ;BX := address of A
        ADD  AX,[BX]      ;AX := integer A+integer B
        POP  BP           ;Restore frame pointer
        RET  4            ;Return, pop 4 bytes
ADD     ENDP
ADDS    ENDS
        END

```

Remember that calling an EXTERN procedure or function at runtime pushes parameters on the stack. An assembly language routine must rely on these pushed parameters being in a certain sequence and format. It must also remove all parameters from the stack before returning.

Assembly language routines must save and restore the BP and DS registers. They must not even modify the SS register. However, the assembly language routines can change the remaining registers (FLAGS, AX, BX, CX, DX, SI, DI, and ES) as necessary.

If the routine is a function, the return value is placed in registers. If the return value is a 1-byte value, it is placed in the AL register, as shown in Figure 8-3. You need not set AH.

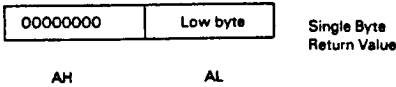


Figure 8-3. 1-Byte Return Value

If the return value is a 2-byte value, the return value is placed in the AX register pair, high byte in AH, and low byte in AL.

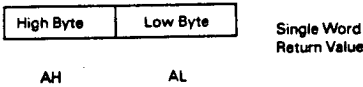


Figure 8-4. 2-Byte Return Value

If the return value is a 4-byte value, the high part (or segment) of the return value is placed in the DX register and the low part (or offset) in the AX register. (This is sometimes shown as DX:AX.) This only applies to INTEGER4 and ADS types.

Since Wang PC Pascal permits a function to retrieve structured values, the return value's size in bytes can be extremely large. Therefore, for all function returns of any real or structured type (REAL4, REAL8, array, record, or set) or of a pointer to a super array type, the compiler allocates its own temporary variable. This occurs even if the size of the return value is 1, 2, or 4 bytes.

The address of this temporary variable is pushed on the stack after all parameters, just before the return address is pushed, as shown in Figure 8-5. (This address is an offset, and therefore only one word is pushed.)

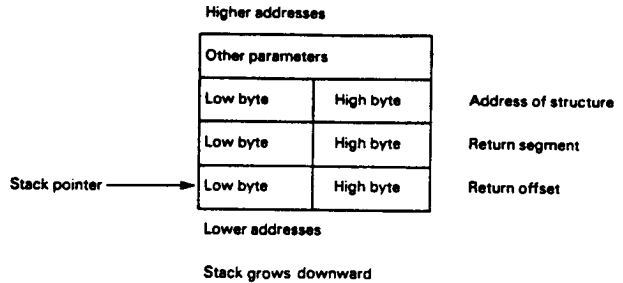


Figure 8-5. 4-Byte Return Value

When this temporary value leaves the function, its address should be placed in the AX register in lieu of the full structure. This address is simply an offset returned in the AX register.

You may wish to pass data using PUBLIC and EXTERN variables instead of parameters. If so, these variable declarations go into a segment named DATA with class name 'DATA', in group DGROUP. It is important that you give the correct segment, class, and group names, as shown in the last example. (See The Wang Professional Computer Assembly Reference Manual for more information on these topics.)

CHAPTER 9

ADVANCED TOPICS

This chapter contains advanced technical information that is of interest primarily to experienced programmers. Because Wang PC FORTRAN and Wang PC Pascal have the same compiler back end and share a common file and runtime system, much of the information that follows refers to both languages. Differences, where they exist, are noted.

9.1 THE STRUCTURE OF THE COMPILER

The compiler is divided into three phases, or passes, each of which performs a specific part of the compilation process. Figure 9-1 illustrates the basic structure of the compiler and its relationship to the files that it reads and writes.

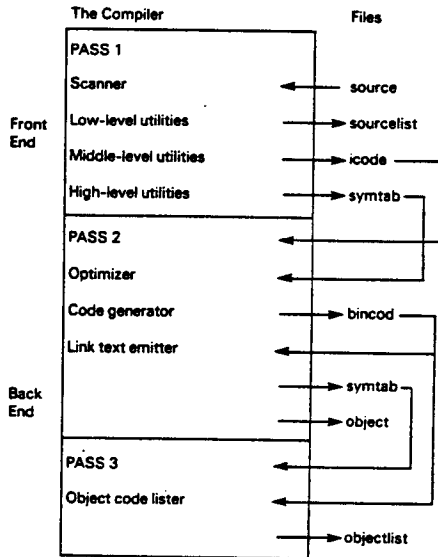


Figure 9-1. The Structure of the Wang PC Pascal Compiler

Pass 1, which normally corresponds to a file named PAS1.EXE, constitutes the front end of the compiler and performs the following actions:

- Reads the source program
- Compiles the source into an intermediate form
- Writes the source listing file
- Writes the symbol table file
- Writes the intermediate code file

Passes 2 and 3 (PAS2.EXE and PAS3.EXE) together make up the back end, which does the following:

- Optimizes the intermediate code
- Generates target code from intermediate code
- Writes and reads the intermediate binary file
- Writes the object (link text) file
- Writes the object listing file

Both the front end and the back end of the compiler are written in Wang PC Pascal, in a source format common to both ISO Pascal and system-level Wang PC Pascal.

All intermediate files contain Wang PC Pascal records. The front and back ends include a common constant and type definition file called PASCOM that defines the intermediate code and symbol table types. The back ends use a similar file for the intermediate binary file definition. Formatted dump programs for all intermediate files and object files are available for special-purpose debugging.

The symbol table record is relatively complex, with a variant for every kind of identifier (assorted data types, variables, procedures and functions). The intermediate code (or Icode) record contains an Icode number, opcode, and up to four arguments; an argument can be the Icode number of another Icode to represent expressions in tree form, or a symbol table reference, constant, length, and the like. The intermediate binary code record contains several variants for absolute code or data bytes, public or external references, label references and definitions, and so on.

9.1.1 The Front End

The Wang PC Pascal front end consists of several parts:

- The scanner
- Low-level utilities

Advanced Topics

- Intermediate-level utilities for identifiers, symbols, Icodes, memory allocation, and type compatibility
- High-level routines for processing procedure and function calls, expressions, statements, and declarations

The front end is driven by recursive descent syntax analysis, using a set of procedures such as EXPR (for expressions), STATEMT (for statements), and TYPEDEC (for type declarations).

The front end maintains a "current" symbol and a "look-ahead" symbol. While not necessary for parsing correct programs, these symbols are useful for error recovery. A procedure that forces the current symbol to one of a set of legal symbols processes syntax errors. If the current symbol is wrong, but the following one is correct, the front end deletes the current symbol. In all cases, the front end inserts the correct symbol if possible. However, common substitution mistakes, such as confusing (=) and (:=), only generate a warning message during compilation.

The scanner is relatively large because it must process metalanguage and produce a listing with error messages, data about variables, and other information for the user.

The front end writes intermediate code to the Icode file on disk as soon as it generates it: there is no reason to keep it in memory. The symbol table is built as a binary tree of identifiers with pointers to semantic records. At the end of each block, the front end writes all new semantic records to the symbol table file. When an error occurs, all writing to intermediate files stops because the code may not be acceptable to the back end. Detecting a warning, rather than an error, does not invalidate the intermediate files.

The front end reads a file called PASKEY to initialize predeclared identifiers such as INTEGER, READ, and MAXINT. PASKEY consists of four sections. The first contains the number of bytes in a file control block and the primitive type identifiers. The second section lists all the intrinsic procedure and function identifiers (those that the front end transforms in special ways). The third section contains constants, types, and external procedures and functions using normal Wang PC Pascal syntax. The fourth contains one or more INTERFACE and USES clauses for predeclared procedures and functions.

9.1.2 The Back End

Two separate passes make up the back end of the compiler; Pass 2 is required, and Pass 3 is optional. Pass 2 produces the object file; Pass 3 produces the object listing.

Pass 2

When the optimizer reads the interpass files, it reads the symbol table for a block first, and then reads the intermediate code for the block. Optimization is performed on each "basic block," or each block of intermediate code up to the first internal or user label or up to a fixed maximum number of Icodes, whichever comes first.

Within this basic block, the optimizer can reorder and condense expressions as long as it preserves the intent of the program(mer). For instance, in the following program fragment, the optimizer need only calculate the array address A [J, K] once.

```
A [J, K] := A [J, K] + 1;
{J := J - 1;}
IF A [J, K] = MAX THEN PUNT;
```

However, if the preceding program fragment is rewritten to include the assignment to J, shown in the fragment as a comment, the optimizer must recalculate the array address in the IF statement. This process is called common subexpression elimination. The optimizer also processes the most complicated parts of expressions first, when more registers for temporary values are available. Several other optimizations occur, such as:

- Constant folding the front end did not do
- Strength reduction (changing multiplications and divisions into shifts when possible)
- Peephole optimization (removing additions of zero, multiplications by one, and changing A := A + 1 to an internal increment memory Icode)

The optimizer works by building a tree out of the intermediate codes for each statement and then transforming the list of statement trees.

Seven internal passes occur for each basic block:

1. Statement tree construction from the Icode stream
2. Preliminary transformations to set address/value flags
3. Length checks and type coercions
4. Constant and address folding, and expression reordering
5. Peephole optimization and strength reduction
6. Machine-dependent transformations
7. Common subexpression elimination

Finally, the optimizer calls the code generator to translate the basic block from tree form to target machine code.

Advanced Topics

The code generator must translate these trees into actual machine code. It uses a series of templates to generate more efficient code for special cases. For example, there is a series of templates for the addition operator. The first template checks for an addition of the constant 1. If this addition is found, the template generates an increment instruction. If the template does not find an addition of 1, then the next template takes control and checks for an addition of any constant. If it finds one, the second template generates an add immediate instruction.

The final template in the series must handle the general case. It moves the operands into registers (by recursively calling the code generator itself), then generates an add register instruction. A series of templates exists for every operation. The code generator must also keep track of register contents and several memory segment addresses (code, static variables, constant data, etc.). In addition, it must also allocate any temporary variables.

The code generator writes a file of binary intermediate code (BINCOD) that contains actual byte values for machine instructions, symbolic references to external routines and variables, and other kinds of data. A final internal pass reads the BINCOD file and writes the object code file.

Pass 3

This short pass reads both the BINCOD file and a version of the symbol table file as updated by the optimizer and code generator. Using the data in these files, it writes the generated code in an assembly-like format.

9.2 AN OVERVIEW OF THE FILE SYSTEM

Because Wang PC Pascal and Wang PC FORTRAN share the same file system, this section includes references to differences between the two wherever they exist. Wang PC Pascal and Wang PC FORTRAN are designed to be connected to existing operating systems easily. The standard interface has two parts:

1. A file control block (FCB) declaration
2. A set of procedures and functions, called Unit U, that Wang PC Pascal or Wang PC FORTRAN call at runtime to perform input and output

This interface supports three access methods: `TERMINAL`, `SEQUENTIAL`, and `DIRECT`.

Each file has an associated FCB. The FCB record type begins with a number of standard fields that are independent of the operating system. Fields that are dependent on the operating system, such as channel numbers, buffers, and other data, follow these standard fields.

The advanced Wang PC Pascal user can access FCB fields directly, as explained in Chapter 16. There is no standard way to access FCB fields within Wang PC FORTRAN.

Both Wang PC Pascal and Wang PC FORTRAN have two special file control blocks that correspond to the keyboard and the screen of your terminal. These two file control blocks are always available. In Wang PC Pascal, they are the predeclared files INPUT and OUTPUT (which you can reassign and generally treat as any other files). In Wang PC FORTRAN, these file control blocks are unit number 0 (or *); Wang PC Pascal and Wang PC FORTRAN access them through a variable TRMVQQ, declared as follows:

```
VAR TRMVQQ: ARRAY [BOOLEAN] OF ADR OF FCBFQQ;
```

The false element references the output file; the true element references the input file.

For Wang PC Pascal files, each FCB ends with the buffer variable that contains the current file component. This means that the length of an FCB in Wang PC Pascal is the length of its fixed portion plus the length of the buffer variable. Wang PC FORTRAN files do not require buffer variables, so all are of a fixed length.

File control blocks always reside in the default data segment, so Wang PC Pascal and Wang PC FORTRAN can reference them with the offset (ADR) addresses instead of the segmented (ADS) addresses.

Wang PC Pascal file variables can occur in static memory, on the stack as local variables, or in the heap as heap variables.

In Wang PC Pascal, generated code initializes file control blocks when the compiler allocates them and closes them when the compiler deallocates them. The Wang PC FORTRAN compiler allocates files during OPEN and deallocates them during CLOSE or at program termination.

The manner of allocation and deallocation depends on the operating system. For example, a fixed number of file "slots" or the routines for Wang PC Pascal heap allocation may be available. Both Wang PC Pascal and Wang PC FORTRAN can create or destroy an FCB, but can never move or copy one.

The Wang PC Pascal Compiler must know enough about an FCB to allocate one. Thus, it needs to know the length of an FCB less the length of its buffer variable. The compiler reads this information from a special file called PASKEY during initialization. The Wang PC FORTRAN Compiler does not allocate files, so it does not need to know the length of an FCB.

Unit U refers to the target operating system interface routines. The file routines specific to Wang PC Pascal are Unit F; the file routines specific to Wang PC FORTRAN are Unit V. Code that either compiler generates, contains calls to Unit F (Wang PC Pascal) or Unit V (Wang PC FORTRAN), which in turn call Unit U routines. This relationship is shown schematically in Figure 9-2.

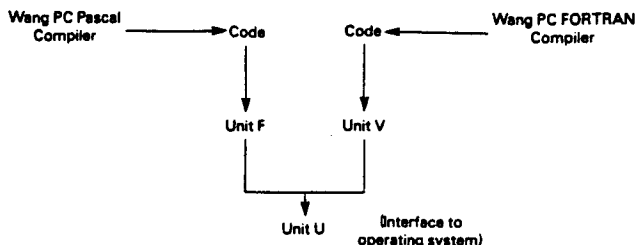


Figure 9-2. The Unit U Interface

The file system uses the following naming convention for public Linker names:

- All Linker globals are six alphabetic characters, ending with QQ. (This helps to avoid conflicts with program global names.)
- The fourth letter indicates a general class, where:
 - a. xxxFQQ is part of the generic Wang PC Pascal file unit.
 - b. xxxVQQ is part of the generic Wang PC FORTRAN file unit.
 - c. xxxUQQ is part of the operating system interface unit.

File system error conditions can occur at the lower Unit U level or at the higher Unit F or V level; the unit may or may not detect an error condition. When a Unit U routine detects an error, it sets an appropriate flag in the FCB and returns to the calling Unit F or V routine. When Unit F or V detects an error or discovers Unit U has detected one, it takes one of two possible actions:

- The unit generates an immediate runtime error message and the program aborts.
- Unit F or V returns to the calling program if you set error trapping (in Wang PC Pascal with the TRAP flag, in Wang PC FORTRAN with the ERR=nnn or IOSTAT=var clauses).

Units F and V will not pass a file with an error condition to a Unit U routine. For some access methods, certain file operations may lead to an undetected error, such as reading past the end of a record (this condition has undefined results). Runtime errors that cause a program abort use the standard error-handling system, which gives the context of the error and provides entry to the target debugging system.

The distributed implementation of the Wang PC Pascal Compiler includes the following three source files:

- FINU contains procedure and function headers for all Unit U routines.
- FINK contains the common FCB declarations for all Wang PC Pascal systems, along with the declaration of the FILEMODES type.
- FINKxx contains the FCB declarations as extended for use in a particular environment. For the Wang PC environment, the name is FINKXM. (These extensions are currently the same for the Wang PC, CP/M(r)-80, and CP/M-86 environments.)

9.3 RUNTIME ARCHITECTURE

The remainder of this chapter describes several topics related to the runtime structure of Wang PC FORTRAN and Wang PC Pascal. It also describes the differences between the languages.

9.3.1 Runtime Routines

Wang PC Pascal and Wang PC FORTRAN runtime entry points and variables conform to the same naming convention: all names are six characters, and the last three are a unit identification letter followed by the letters QQ. Table 9-1 shows the current unit identifier suffixes.

Table 9-1. Unit Identifier Suffixes

Suffix	Unit Function	Suffix	Unit Function
AQQ	Complex real	NQQ	Long integer
BQQ	Compile time utilities	OQQ	Other miscellaneous routines
CQQ	Encode, decode	PQQ	Pcode interpreter
DQQ	Double-precision real	QQQ	Reserved
EQQ	Error-handling	RQQ	Real (single-precision)
FQQ	Wang PC Pascal file system	SQQ	Set operations
GQQ	Generated code helpers	TQQ	Reserved
HQQ	Heap allocator	UQQ	Operating system file system
IQQ	Generated code helpers	VQQ	Wang PC FORTRAN file system
JQQ	Generated code helpers	WQQ	Reserved
KQQ	FCB definition	XQQ	Initialize/terminate
LQQ	STRING, LSTRING	YQQ	Special utilities
MQQ	Reserved	ZQQ	Reserved

9.3.2 Memory Organization

Memory on the 8086 is divided into segments, each containing up to 64 KB. The relocatable object format and the Linker also put segments into classes and groups. The 8086 loads all segments with the same class name are next to each other. All segments with the same group name must reside in one area up to 64 KB long, so one 8086 segment register can access all segments in a group.

Wang PC FORTRAN and Wang PC Pascal both define a single group, named DGROUP, which the DS or SS segment registers address. Normally, DS and SS contain the same value, although DS may change temporarily to some other segment and change back again. SS does not change; its segment registers always contain abstract "segment values" and the contents remain the same. This provides compatibility with the Intel(R) 8086 processor. Long addresses, such as Wang PC Pascal ADS variables or Wang PC FORTRAN named common blocks, use the ES segment register for addressing.

DGROUP contains memory for all static variables, constants that reside in memory, the stack, the heap, Wang PC FORTRAN blank common, and segmented addresses of Wang PC FORTRAN named common blocks. The named common blocks themselves reside in their own segments, not in DGROUP.

Memory allocation in DGROUP occurs from the top down; that is, the highest addressed byte has DGROUP offset 65535, and the lowest allocated byte has some positive offset. This allocation means offset zero in DGROUP may address a byte in the code portion in memory, in the operating system below the code, or even below absolute memory address zero (in the latter case, the values in DS and SS are "negative").

DGROUP has two parts:

- A variable-length lower portion that contains the heap and the stack
- A fixed-length upper portion that contains static variables, constants, blank common, and named common addresses

During program initialization (in ENTX6L), the fixed upper portion moves upward as much as possible to make room for the lower portion. If there is enough memory, DGROUP expands to the full 64 KB; if there is not enough for this, it expands as much as possible.

Figure 9-3 illustrates memory organization. The paragraphs after the figure describe memory contents, starting at the bottom (address zero), when a Wang PC FORTRAN or Wang PC Pascal program is running. Addresses are shown in segment:offset form.

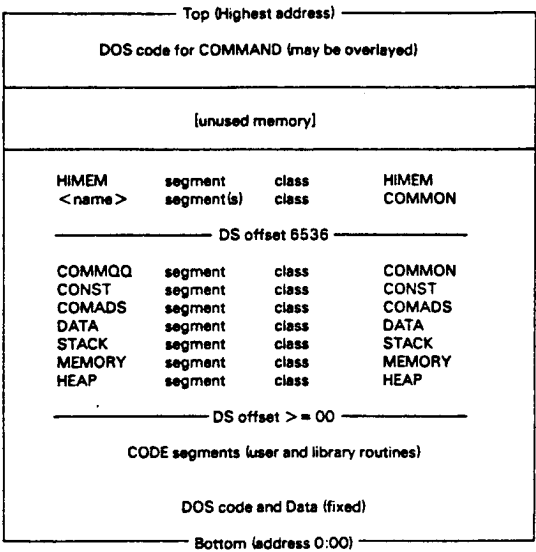


Figure 9-3. Memory Organization

0000:0000 The beginning of memory on an 8086 system contains interrupt vectors, which are segmented addresses. Usually, the first 32 to 64 are reserved for the operating system. The resident portion of the operating system follows these vectors.

The operating system provides for loading additional code above it, which remains resident and is considered part of the operating system as well. Examples of resident additional code are special device drivers for peripherals, a print spooler, or the debugger.

BASE:0000 Here, BASE means the starting location for loaded programs, sometimes called the transient program area. When you invoke a Wang PC FORTRAN or Wang PC Pascal program, loading begins here. The beginning of your program contains the code portion, with one or more code segments. These code segments are in the same order as the object modules given to the Linker, followed by object modules loaded from libraries.

DGROUP:LO

The DGROUP data area contains the following:

Segment	Class	Description
HEAP	MEMORY	Pointer variables, some files (not used, Intel-compatible)
MEMORY	MEMORY	
STACK	STACK	Frame variables and data
DATA	DATA	Static variables
COMADS	COMADS	Addresses of named commons
CONST	CONST	Constant data
COMMQQ	COMMON	FORTRAN blank common

The stack and the heap grow toward each other, the stack downward and the heap upward.

DGROUP:TOP

TOP means 64 KB (4K paragraphs) above DGROUP:0000 (that is, just past the end of DGROUP). Wang PC FORTRAN named common blocks start here. Each common block has a segment name as declared in the Wang PC FORTRAN program as the common block name, and the class name COMMON. Each named common has one segmented (ADS) address in the COMADS segment in DGROUP. All references to common block component variables use offsets from this address.

HIMEM:0000

The segment named HIMEM (class HIMEM) gives the highest used location in the program. The segment itself contains no data, but the initialization process uses its address. Available memory starts here; Wang PC Pascal ADS variables can access this memory.

COMMAND

The DOS Command Processor (the part that performs COPY, DIR, and other resident commands) is in the highest possible location in memory. Your Wang PC FORTRAN or Wang PC Pascal program may need this memory area in order to run. If so, program data overwrites the command processor. When your program finishes, the file COMMAND.COM on the default drive reloads the command processor.

In some circumstances, the check may cause a message to appear on your screen that tells you to insert a disk that contains the file COMMAND.COM. You can avoid this delay by making sure that COMMAND.COM is on the disk in the default drive when the program ends.

9.3.3 Initialization and Termination

Every executable file contains one, and only one, starting address. As a rule, when Wang PC Pascal or Wang PC FORTRAN object modules are involved, this starting address is at the entry point BEGXQQ in the module ENTX. Other letters may be appended to the name ENTX, but the name of the module always begins with the four letters ENTX. A Wang PC Pascal or Wang PC FORTRAN program (as opposed to a module or implementation) has a starting address at the entry point ENTGQQ. BEGXQQ calls ENTGQQ.

The following discussion assumes that you have loaded and executed a Wang PC Pascal or Wang PC FORTRAN main program along with other object modules. However, you can also link a main program in assembly or some other language with other object modules in Wang PC Pascal or Wang PC FORTRAN. In this case, modules other than ENTX perform some of the initialization and termination.

When program execution begins, several levels of initialization must occur. The levels, in the order in which they occur, are:

- Machine-oriented initialization
- Runtime initialization
- Program and unit initialization

The general scheme is shown in Figure 9-4.

The entry point of a Wang PC Pascal load module is the routine BEGXQQ, in the module ENTX. (The module may also be called ENTX8, ENTX6M, etc.). BEGXQQ does the following:

1. Moves constant and static variables upward (as described in Section 9.3.2) creating a gap for the stack and the heap. It sets the stack pointer to the top of this area. The initial stack pointer is put into PUBLIC variable STKBQQ and restores the stack pointer after an interprocedure GOTO to the main program.
2. Sets the frame pointer to zero.
3. Initializes a number of PUBLIC variables to zero or NIL. These include:
 - RESEQQ, machine error context
 - CSXEQQ, source error context list header
 - PNUXQQ, initialized unit list header
 - HDRFQQ, Wang PC Pascal open file list header
 - HDRVQQ, Wang PC FORTRAN open file list header
4. Sets machine-dependent registers, flags, and other values.
5. Sets the heap control variables. It sets BEGHQQ and CURHQQ to the lowest address for the heap, and sets the word at this address to a heap block header for a free block the length of the initial heap. It sets ENDHQQ to the address of the first word after the heap. As stack and the heap grow together, BEGXQQ sets the PUBLIC variable STKHQQ to the lowest legal stack address (ENDHQQ, plus a safety gap).

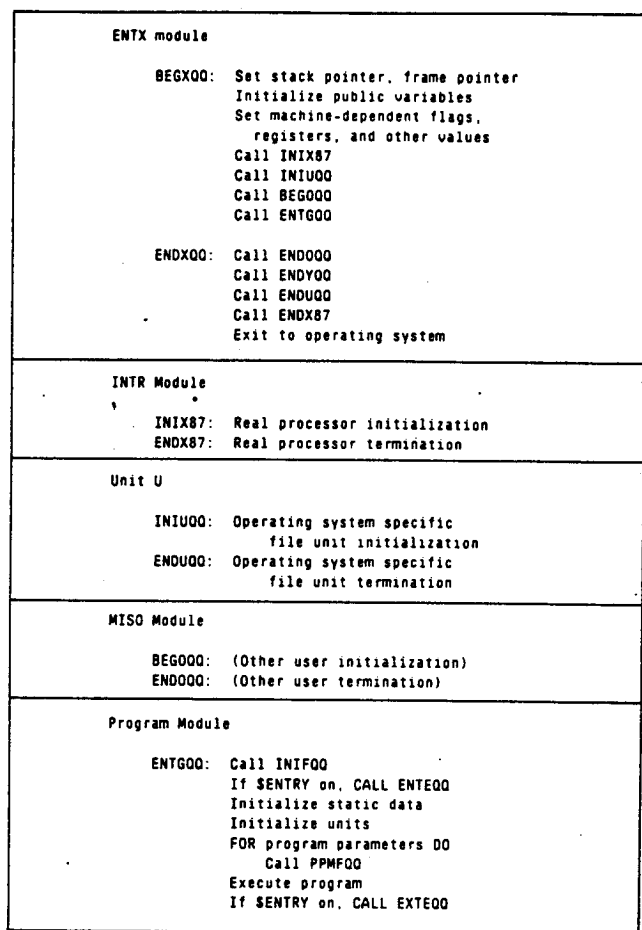


Figure 9-4. Wang PC Pascal Program Structure

6. Calls INIX87, the real processor initializer. This routine initializes an 8087 or sets 8087 emulator interrupt vectors, as appropriate.
7. Calls INIUQQ, the file unit initializer specific to the operating system. If you do not want to load a file unit, you must instead load a dummy INIUQQ routine that merely returns.
8. Calls BEGOQQ, the escape initializer. A normal load module includes an empty BEGOQQ that only returns. However, this call provides an escape mechanism for any other initialization. For example, it could initialize tables for an interrupt-driven profiler or a runtime debugger.
9. Calls ENTGQQ, the entry point of your Wang PC Pascal program.

Your main program continues the initialization process. First, the language-specific file system is called. INIFQQ for Wang PC Pascal or INIVQQ for Wang PC FORTRAN. Both procedures lack parameters.

If the main program is in Wang PC Pascal and uses Wang PC FORTRAN file routines, you must call INIVQQ to initialize the Wang PC FORTRAN file system. If the main program is in Wang PC FORTRAN and uses Wang PC Pascal file routines, you must call INIFQQ to initialize the Wang PC Pascal file system.

Wang PC Pascal main programs automatically call INIFQQ; Wang PC FORTRAN main programs automatically call INIVQQ. To avoid loading the file system, you must provide an empty procedure to satisfy one or both of these calls.

After file initialization, the main program calls ENTEQQ to set the source error context (but only if \$ENTRY is on during compilation). Next, each file at the program level gets an initialization call to NEWFQQ.

Unit initialization follows static data initialization. Every USES clause in the source, including those in interfaces, generates a call to the initialization code for the unit.

Units may or may not contain initialization code. If the interface contains a trailing pair of BEGIN and END statements, it assumes that initialization code is present in the implementation. The interface then initializes units in the order in which it encounters the USES clauses.

Finally, the compiler reads or initializes any program parameters, and your program begins. The compiler sets program parameters in one of a number of ways. In general, except for INPUT and OUTPUT, PPMFQQ sets the parameter's string value as the next line in the file INPUT. Then, one of the READFN routines "reads" and decodes the value, assigning it to the parameter. PPMFQQ then uses the parameter's identifier as a prompt. PPMFQQ first calls PPMUQQ to get the text of any command line parameter or other parameters specific to the operating system. If PPMUQQ returns an error, then PPMFQQ does the prompting and reads the response directly.

User unit initialization is much like user program initialization. The following actions occur:

- Error context initialization if the \$ENTRY metaccommand was on during compilation
- Variable (file) initialization
- Unit initialization for USES clause
- User's unit initialization

Calls to initialize a unit may come from more than one unit. The unit interface has a version number. Each initialization call must check that the version number that was in effect for previous compilations of the unit is the same as the version number that was in effect for the first compilation of the unit implementation itself.

Except for this, unit initialization calls after the first one should have no effect; a unit's initialization code executes only once. Code at the start of the body of the unit handles both version-number checking and single, initial-code execution. This has the effect of:

```
IF INUXQQ (useversion, ownversion, initrec, unitid)
THEN RETURN
```

The compilant using the interface passes the interface version number as a value parameter to the implementation initialization code, and as useversion to INUXQQ. The compilant then passes the interface version number in the implementation itself as ownversion to INUXQQ. INUXQQ generates an error if the two are unequal.

INUXQQ also maintains a list of initialized units. INUXQQ returns true if the unit is found in the list. If not, it puts the unit in the list and returns false. The list header is PINUXQQ. A list entry passed to INUXQQ as initrec is initialized to the address of the unit's identifier (unitid), with a pointer to the next entry.

User modules (and uninitialized implementations of units) may have initialization code that is much like a program and unit implementation's initialization code, but without user initialization code or INUXQQ calls.

A module cannot issue the initialization call for a module or uninitialized unit automatically. During compilation, a warning occurs if an initialization call is required (that is, if any files are declared or if USES clauses exist). To initialize a module, declare the module name as an external procedure and call it at the beginning of the program.

Program Termination

Program termination occurs in one of three ways:

1. The program may terminate normally, in which case the main program returns to BEGXQQ, at the location named ENDXQQ.
2. The program may abort because of an error condition, either with a user call to ABORT or a runtime call to an error handling routine. In either case, the program passes an error message, error code, and error status to EMSEQQ, which does whatever error handling it can and calls ENDXQQ.
3. An external procedure can declare ENDXQQ and call it directly.

ENDXQQ first calls ENDOQQ, the escape terminator, which normally returns to ENDXQQ. Then ENDXQQ calls ENDYQQ, the generic file system terminator. ENDYQQ closes all open Wang PC Pascal and Wang PC FORTRAN files, using the file list headers HDRFQQ and HDRVQQ. ENDXQQ calls ENDUQQ, the file unit terminator that is operating-system-specific. Finally, ENDXQQ calls ENDX87 to terminate the real number processor (8087 or emulator). As with INTUQQ, INIFQQ, and INIVQQ, if your program requires no file handling, you must declare empty procedures that have no parameters for ENDYQQ and ENDUQQ. The main initialization and termination routines are in module ENTX. Procedures for BEGOQQ and ENDOQQ are in module MISO. ENDYQQ is in module ENDY.

9.3.4 Error-Handling

Programs and routines can detect runtime errors in one of four ways:

1. The user program calls EMSEQQ (ABORT).
2. A runtime routine calls EMSEQQ.
3. An error-checking routine in the error module calls EMSEQQ.
4. An internal helper routine calls an error message routine in the error unit that in turn calls EMSEQQ.

Handling an error detected at runtime usually involves identifying the type and location of the error and then terminating the program. The error type consists of a message, an error number, and an error status. In Wang PC Pascal and Wang PC FORTRAN, the message describes the error, and the number refers you to further information (refer to Appendix H). In Wang PC FORTRAN, the error status value has no significance for the user. In Wang PC Pascal, the error status value is undefined, although for file system errors it may be an operating system return code. Table 9-2 shows the general scheme for error code numbering.

Advanced Topics

Table 9-2. Error Code Classification

Range	Classification
1- 999	Reserved for user ABORT calls
1000-1099	Unit U file system errors
1100-1199	Unit F file system errors
1200-1299	Unit V file system errors
1300-1999	Reserved
2000-2049	Heap, stack, memory
2050-2099	Ordinal and long integer arithmetic
2100-2149	Real and double real arithmetic
2150-2199	Structures, sets and strings
2200-2399	Reserved
2400-2449	Pcode interpreter
2450-2499	Other internal errors
2500-2999	Reserved

An error location has two parts: the machine error context, and the source program context. The machine error context is the program counter, stack pointer, and frame pointer at the point of the error. The program counter is always the address that follows a call to a runtime routine (such as a return address). The source program context is optional; metacommands control it. If the \$ENTRY metacommand is on, the program context consists of:

- The source file name of the compilant that contains the error
- The name of the routine in which the error occurred (program, unit, module, procedure, or function)
- The line number of the routine in the listing file
- The page number of the routine in the listing file

If the \$LINE metacommand is also on, the line number of the statement that contains the error is also given. Setting \$LINE also sets \$ENTRY.

Machine Error Context

Setting the \$RUNTIME metacommand causes compilation of runtime routines. This metacommand generates special calls at the entry and exit points of each runtime routine. The entry call saves the context at the point where the user program calls a runtime routine. This context consists of the frame pointer, stack pointer, and program counter. As a consequence of this saving of context, if an error occurs in a runtime routine, the error location is always in the user program. This is true even if runtime routines call other runtime routines. The exit call restores the context. The runtime entry helper, BRTEQQ, uses the runtime values shown in Table 9-3.

Table 9-3. Runtime Values in BRTEQQ

Value	Description
RESEQQ	Stack pointer
REFEQQ	Frame pointer
REPEQQ	Program counter offset
RECEQQ	Program counter segment

BRTEQQ first examines RESEQQ. If this value is not zero, the current runtime routine simply returns. If RESEQQ is zero, however, BRTEQQ must save the error context. BRTEQQ determines the caller's stack pointer from the current frame pointer and stores it in RESEQQ. BRTEQQ then determines the address of the caller's saved frame pointer and return address (program counter) in the frame and saves the frame pointer in REFQQ. BRTEQQ finally saves the caller's program counter (BRTEQQ's caller's return address): the offset in REPEQQ and the segment (if any) in RECEQQ.

The runtime exit helper, ERTEQQ, has no parameters. It determines the caller's stack pointer (again, from the frame pointer) and compares it against RESEQQ. If these values are equal, the original runtime routine your program calls is returning, so ERTEQQ sets RESEQQ back to zero.

EMSEQQ uses RESEQQ, REFQQ, REPEQQ, and RECEQQ to display the machine error context.

Source Error Context

Giving the source error context involves extra overhead because the object code must include source location data in some form. Currently, calls set the current source context as it occurs. These calls can also break program execution as part of the debug process. The overhead of source location data, especially line number calls, can be significant. Routine entry and exit calls, while they require more overhead, are much less frequent, so the overhead is less.

The procedure entry call to ENTEQQ passes two VAR parameters. The first is an LSTRING that contains the source file name. The second is a record that contains:

- The line number of the procedure (a WORD)
- The page number of the procedure (a WORD)
- The procedure or function identifier (an LSTRING)

The file name is that of the compiland source (such as the main source file name, not the names of any \$INCLUDE files). The procedure identifier is the full identifier the source uses, not the Linker name. If one name appears in an INTERFACE and another in a USES clause, the source uses the USES identifier. The line and page are those the procedure header designates.

Advanced Topics

Entry and exit calls occur in the main program, unit initialization, and module initialization, in which case the identifier is the program, unit, or module.

The procedure exit call to EXTEQQ does not pass any parameters. It pops the current source routine context off a stack in the heap.

The line number call to LNTEQQ passes a line number as a value parameter. The current line number resides in the PUBLIC variable CLNEQQ. Because the current routine is always available (because \$LINE implies \$ENTRY), the compilant source file name and routine that contain the line are available along with the line number. Line number calls occur just before the code in the first statement on a source line. The statement can, of course, be part of a larger statement. The \$LINE+ metaccommand should occur at least a couple of symbols before the start of the first statement intended for a line number call (\$LINE- also takes effect early).

Most of the error-handling routines are in modules ERRE and PASE. The source error context entry points ENTEQQ, EXTEQQ, and LNTEQQ are in the debug module, DEBE.

Part 2

The Wang Professional Computer
Pascal Syntax

CHAPTER 10 LANGUAGE OVERVIEW

This chapter contains a summary description of Wang PC Pascal language elements. Each of the remaining chapters of the manual discusses these elements in detail, from the smallest element (notation) to the largest (metacommands).

Many of the explanations of elements of notation refer to the extend-level of Wang PC Pascal. Extend-level elements are those that are not part of standard Pascal, but are available in Wang PC Pascal. Section 23.1.5 and Appendices B, E, and F contain listings of extend-level elements.

10.1 METACOMMANDS

The Wang PC Pascal metacommands provide a control language for the Wang PC Pascal compiler. The metacommands let you specify options that affect the overall operation of a compilation. For example, you can conditionally compile different source files, generate a listing file, or enable or disable runtime error-checking code.

You must insert metacommands inside comment statements. All of the metacommands begin with a dollar sign (\$). You can also give metacommands as switches when you enter the compiler.

Although most implementations of Pascal have some type of compiler control, the Wang PC Pascal metacommands are not part of standard Pascal and hence are not part of other Pascal implementations.

Chapter 26 and Appendix G explain each available Wang PC metacommand and its function.

10.2 PROGRAMS AND COMPILABLE PARTS OF PROGRAMS

Compilands are compilable programs and parts of programs. Wang PC Pascal compilands include programs, modules, and units. You can compile modules and units separately and then, later, link them to a program without having to recompile the module or unit.

The fundamental unit of compilation is a program. A program has three parts:

1. The program heading identifies the program and gives a list of program parameters.
2. The declaration section follows the program heading and contains declarations of labels, constants, types, variables, functions, and procedures. Some of these declarations are optional.
3. The body follows all declarations. It is enclosed by the reserved words BEGIN and END and ends with a period. The period signals the compiler that it has reached the end of the source file.

The following program illustrates this three-part structure:

```
{Program heading}
PROGRAM FRIDAY (INPUT,OUTPUT);

{Declaration section}
LABEL 1;
CONST DAYS_IN_WEEK = 7;
TYPE KEYBOARD_INPUT = CHAR;
VAR KEYIN: KEYBOARD_INPUT;

{Program body}
BEGIN
  WRITE('IS TODAY FRIDAY? ');
1: READLN(KEYIN);
  CASE KEYIN OF
    'Y', 'y' : WRITELN('It's Friday.');
```

```
    'N', 'n' : WRITELN('It's not Friday.');
```

```
  OTHERWISE
    WRITELN('Enter Y or N.');
```

```
    WRITE('Please re-enter: ');
```

```
    GOTO 1
```

```
  END
```

```
END.
```

This three-part structure (heading, declaration section, body) is used throughout the Pascal language. Procedures, functions, modules, and units are all similar in structure to a program.

Modules are compilands that contain the declaration of variables, constants, types, procedures, and functions, but no program statements. You can compile a module separately and later link it to a program, but you cannot execute a module by itself. An example of a module follows:

```

{Module heading}
MODULE MODPART;

{Declaration section}
CONST PI = 3.14;

PROCEDURE PARTA;
BEGIN
    WRITELN ('parta')
END;

{Body}
END.

```

A module, like a program, ends with a period. Unlike a program, a module contains no program statements.

A unit is a compiland that consists of two sections: an interface and an implementation. The interface contains the information that lets you connect a unit to other units, modules, and programs. Like a module, you can compile an implementation separately and later link it to the rest of the program. An example of a unit follows:

```

{Heading for interface}
INTERFACE;
UNIT MUSIC (SING, TOP);

{Declarations for interface}
VAR TOP : INTEGER;
PROCEDURE SING;

{Body of interface}
BEGIN
END;

{Heading for implementation}
IMPLEMENTATION OF MUSIC;

{Declaration for implementation}
PROCEDURE SING;
BEGIN
    FOR I := 1 TO TOP DO
        BEGIN
            WRITE ('FA '); WRITELN ('LA LA')
        END
    END;

{Body of implementation}
BEGIN
    TOP := 5
END.

```

A unit, like a program or a module, ends with a period.

Modules and units let you develop large structured programs made up of several parts. This practice is helpful in the following situations:

- If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
- If a program is large and recompiling the entire source file is time-consuming, breaking the program into parts saves compilation time.
- If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs that use the routines.

See Chapter 25 for a complete discussion of programs, modules, and units.

10.3 PROCEDURES AND FUNCTIONS

Procedures and functions act as subprograms that execute under the supervision of a main program. However, unlike programs, procedures and functions can reside within each other and can even call themselves. Furthermore, they have sophisticated parameter-passing capabilities that programs lack.

You invoke procedures as statements; you invoke functions in expressions wherever values are called for. A procedure declaration, like a program, has a heading, a declaration section, and a body. An example of a procedure declaration follows:

```
{Heading}
PROCEDURE COUNT_TO(NUM : INTEGER);

{Declaration section}
VAR I : INTEGER;

{Body}
BEGIN
    FOR I := 1 TO NUM DO WRITELN (I)
END;
```

A function is a procedure that returns a value of a particular type; hence, a function declaration must indicate the type of the return value. An example of a function declaration follows:

```
{Heading}
FUNCTION ADD (VAL1, VAL2 : INTEGER): INTEGER;

{Declaration section empty}

{Body}
BEGIN
    ADD := VAL1 + VAL2
END;
```

Procedures and functions look somewhat different from programs, in that their parameters have types and other options. Like the body of a program, the body of a procedure or a function is enclosed by the reserved words BEGIN and END; however, a semicolon rather than a period follows END.

Declaring a procedure or function is not the same as using it in a program. For example, the procedure and function declared above might actually appear in a program as follows:

```
TARGET_NUMBER := ADD (5, 6);  
COUNT_TO (TARGET_NUMBER);
```

See Chapter 22 for a complete discussion of procedures and functions. See Chapters 23 and 24 for a discussion of predeclared Wang PC Pascal procedures and functions.

10.4 STATEMENTS

Statements perform actions, such as computing, assigning, altering the flow of control, and reading and writing files. Statements can appear in the bodies of programs, procedures, and functions. The system executes statements as a program runs. Wang PC Pascal statements perform the actions shown in Table 10-1. See Chapter 21 for a detailed discussion of each of these statements.

Table 10-1. Summary of Wang PC Pascal Statements

Statement	Purpose
Assignment	Replaces the current value of a variable with a new value.
BREAK	Leaves the loop currently executing.
CASE	Allows for the selection of one action from a choice of many, based on the value of an expression.
CYCLE	Starts the next iteration of a loop.
FOR	Executes a statement repeatedly while a progression of values is assigned to a control variable.
GOTO	Continues processing at another part of the program.
IF	Together with THEN and ELSE, allows for conditional execution of a statement.
Procedure call	Invokes a procedure with actual parameter values.
REPEAT	Repeats a sequence of statements one or more times until a Boolean expression becomes true.
RETURN	Leaves the current procedure, function, program, or implementation.
WHILE	Repeats a statement until a Boolean expression becomes false.
WITH	Opens the scope of a statement to include the fields of one or more records, so that you can refer to the fields directly.

10.5 EXPRESSIONS

An expression is a formula for computing a value. It consists of a sequence of operators (that tell the compiler to perform an action) and operands (the value on which the compiler performs the operation). Operands can contain function invocations, variables, constants, or even other expressions. In the following expression, plus (+) is an operator, while A and B are operands:

$$A + B$$

There are three basic kinds of expressions:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression.
2. Boolean expressions perform logical and comparison operations with Boolean results.
3. Set expressions perform combining and comparison operations on sets, with Boolean or set results.

Expressions always return values of a specific type. For instance, if A, B, C, and D are all real variables, the following expression returns a real result:

$$A * B + (C / D) + 12.3$$

Expressions can also include function designators:

$$\text{ADDREAL}(2, 3) + (C / D)$$

ADDREAL is a function that was previously declared in a program. It has two real value parameters, which it adds together to obtain a total. This total is the return value of the function, which it then adds to (C / D).

Expressions are not statements, but they can be components of statements. In the following example, the entire line is a statement; only the portion after the equals sign is an expression:

$$X := 2 / 3 + A * B$$

See Chapter 20 for a detailed discussion of expressions.

10.6 VARIABLES

A variable is a value that changes during the course of a program. Every variable must be of a specific data type.

After you declare a variable in the heading or declaration section of a compilant, procedure, or function, you can:

- Initialize it in the VALUE section of a program
- Assign it a value with an assignment statement
- Pass it as a parameter to a procedure or function
- Use it in an expression

The VALUE section is a Wang PC Pascal feature that applies only to statically-allocated variables (variables with a fixed address in memory). First, you declare the variables, as shown in the following example:

```
VAR I, J, K, L : INTEGER;
```

Then you assign them initial values in the VALUE section:

```
VALUE I := 1; J := 2; K := 3; L := 4;
```

Later, in statements, you can assign the variables to and use them as operands in expressions:

```
I := J + K + L;
J := 1 + 2 + 3;
K := (J * K) + 9 + (L DIV J);
```

See Chapter 19 for a complete discussion of variables.

10.7 CONSTANTS

A constant is a value that does not change during the course of a program. At the standard level, a constant can be:

- A number, such as 1.234 and 100
- A string enclosed in single quotation marks, such as 'Miracle' or 'A1207'
- A constant identifier that is a synonym for a numeric or string constant

Constants are closely related to the concepts of variables and types. Variables are all of some type; types, in turn, designate a range of assumable values. These values, ultimately, are all constants.

You declare constant identifiers in the CONST section of a compiland, procedure, or function

```
CONST REAL_CONST = 1.234;
      MAX_VAL    = 100;
      TITLE      = 'Pascal';
```

Because the order of declarations is flexible in Wang PC Pascal, you can declare constants as often as necessary anywhere in the declaration section of a compilable part of a program. There can also be more than one constant section in a Wang PC Pascal program.

Structured constants and constant expressions are two powerful extensions in Wang PC Pascal. Examples of these are:

1. VECTOR, in the following example, is an array constant:

```
CONST VECTOR = VECTORTYPE (1,2,3,4,5);
```

2. MAXVAL, in the following example, is a constant expression (A, B, C, and D must also be constants):

```
CONST MAXVAL = A * (B DIV C) + D - 5;
```

See Chapter 18 for a complete discussion of these and other aspects of constants.

10.8 TYPES

Much of Pascal's power and flexibility lies in its data typing capability. Although many data types are available, they fall into three broad categories:

- Simple -- a simple data type represents a single value. The simple types include the following:

INTEGER	enumerated
WORD	subrange
CHAR	REAL
BOOLEAN	INTEGER4

- Structured -- a structured type represents a collection of values. The structured types include the following:

```
ARRAY
RECORD
SET
FILE
```

- Reference -- reference types allow recursive definition of types.

All variables in Pascal must have a data type. A type is either predeclared (for example, INTEGER and REAL) or defined in the declaration section of a program. The following sample type declaration creates a type that can store information about a student:

```

TYPE
  STUDENT = RECORD
    AGE      : 5..18;
    SEX      : (MALE, FEMALE);
    GRADE    : INTEGER;
    GRADE_PT : REAL;
    SCHEDULE : ARRAY [1..10] OF CLASSES
  END;

```

For a detailed discussion of data types, see Chapters 13 through 17.

10.9 IDENTIFIERS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers. Constants, type, and variables do not require identifiers, but it is useful if they have them.

You make up most of the identifiers in a program and assign them meaning in declarations. Other identifiers are the names of variables, data types, procedures, and functions that are built into the language; you do not need to declare these.

An identifier must begin with a letter (A through Z and a through z). Any number of letters, digits (0 through 9), or underscore characters can follow the initial letter. The compiler ignores the case of letters, so that A and a are equivalent.

The underscore is significant in Wang PC Pascal. Thus, the following are not identical:

FOREST

FOR_EST

The only restriction on identifiers is that you must not choose a Pascal reserved word (see Section 11.3.3 for a discussion of reserved words; see Appendix E for a complete list). Furthermore, most compilers have some restriction either on the absolute length of an identifier or on the number of characters that are considered significant. See Appendix A for any limitations in Wang PC Pascal.

See Chapter 12 for a complete discussion of identifiers in Wang PC Pascal.

10.10 NOTATION

The basis of all Pascal programs is an irreducible set of symbols with which you create the higher syntactic components of the language. The underlying notation is the ASCII character set, divided into the following syntactic groups:

- Identifiers -- the names given to individual instances of components of the language.
- Separators -- characters that delimit adjacent numbers, reserved words, and identifiers.
- Special symbols -- punctuation, operators, and reserved words.
- Unused characters -- The Wang PC Pascal character set does not include some characters that are, nevertheless, available for use in a comment or string literal (refer to Section 11.4).

A good understanding of this notation will increase your productivity by reducing the number of syntax errors in a program. Chapter 11 provides a detailed discussion of Wang PC Pascal notation.

CHAPTER 11 PASCAL NOTATION

All components of the Wang PC Pascal language are constructed from the standard ASCII character set. Characters make up lines, each of which is separated by a character specific to your operating system. Lines make up files.

Within a line, individual characters or groups of characters fall into one (or more) of four broad categories:

1. Components of identifiers
2. Separators
3. Special symbols
4. Unused characters

11.1 COMPONENTS OF IDENTIFIERS

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. The use of identifiers is described thoroughly in Chapter 12; this section discusses only how to construct them.

Identifiers must begin with a letter. Subsequent components can include letters, digits, and underscore characters. Although, in theory, there is no limit on the number of characters in an identifier, most implementations restrict the length in some way. See Appendix A for any limitations that apply to the Wang PC.

11.1.1 Letters

Only the uppercase letters A through Z are significant in identifiers. You can use lowercase letters for identifiers in a source program. However, the Wang PC Pascal compiler converts all lowercase letters in identifiers to the corresponding uppercase letters.

Letters in string literals can be either uppercase or lowercase; the difference is significant to the compiler (that is, the letter A has a different meaning than the letter a). No mapping of lowercase to uppercase occurs in either comments or string literals.

11.1.2 Digits

Digits in Pascal are the numbers zero through nine. Digits can occur in identifiers (for example, AS129M) or in numeric constants (for example, 1.23 and 456).

11.1.3 The Underscore Character

The underscore (`_`) is the only nonalphanumeric character allowed in identifiers. The underscore is significant in Wang PC Pascal; use it as a space to improve readability. For example, the identifiers in the right column below are easier to read than those in the left column:

POWEROFTEN
MYDOGMAUDE

POWER_OF_TEN
MY_DOG_MAUDE

11.2 SEPARATORS

Separators delimit adjacent numbers, reserved words, and identifiers, none of which should have separators embedded within them. A separator can be any of the following:

- Space character
- Tab character
- Form feed character
- New line marker
- Comment

Comments in standard Pascal take one of these forms:

{This is a comment, enclosed in braces.}

(*This is an alternate form of comment.*)

You can generate a left brace on the Wang PC keyboard by pressing the 2ND, SHIFT, and bracket (`[`) keys simultaneously. To obtain a right brace, press the 2ND and bracket keys simultaneously. In either of the above forms, comments can span more than one line.

Pascal Notation

At the extend level, Wang PC Pascal also allows comments that begin with an exclamation point:

! The rest of this line is a comment.

For comments in this form, the new line character delimits the comment.

Nested comments are permitted in Wang PC Pascal, as long as each level has different delimiters. Thus, when a comment begins, the compiler ignores succeeding text until it finds the matching end-of-comment.

Always use separators between identifiers and numbers. If you fail to do so, the compiler generally issues an error or warning message.

In a few cases, the Wang PC Pascal compiler accepts a missing separator without generating an error message. For example, at extend level, the compiler accepts 100MOD#127 as 100 MOD #127, where #127 is a hexadecimal number. However, the compiler assumes that 100MOD127 is 100 followed by the identifier MOD127.

11.3 SPECIAL SYMBOLS

Special symbols fall into three categories: punctuation, operators, and reserved words.

11.3.1 Punctuation

Punctuation in Wang PC Pascal serves a variety of purposes, including those shown in Table 11-1.

11.3.2 Operators

Operators are a form of punctuation that indicate some operation to be performed. Some are alphabetic, others are one or two nonalphanumeric characters. Any operators that consist of more than one character must not have a separator between characters.

The operators that consist only of nonalphabetic characters are:

+ - * / > < = <> <= >=

Table 11-1. Summary of Punctuation in Wang PC Pascal

Symbol	Purpose
{ }	Braces delimit comments.
[]	Brackets delimit array indices, sets, and attributes. They can also replace the reserved words BEGIN and END in a program.
()	Parentheses delimit expressions, parameter lists, and program parameters.
'	Single quotation marks enclose string literals.
:=	The colon-equals symbol assigns values to variables in assignment statements and in VALUE sections.
;	The semicolon separates statements and declarations.
:	The colon separates variables from types and labels from statements.
=	The equals sign separates identifiers and type clauses in a TYPE section.
,	The comma separates the components of lists.
..	The double period denotes a subrange.
.	The period designates the end of a program, indicates the fractional part of a real number, and also delimits fields in a record.
↑	The up arrow denotes the value a reference value points to.
#	The number sign denotes nondecimal numbers.
\$	The dollar sign prefixes metacommands.

Some operators (for example, NOT and DIV) are reserved words instead of nonalphabetic characters.

See Chapter 20 for a complete list of the nonalphabetic operators and discussion of the use of operators in expressions.

Pascal Notation

11.3.3 Reserved Words

Reserved words are a fixed part of the Wang PC Pascal language. They include, for example, statement names (such as BREAK) and words like BEGIN and END that bracket the main body of a program.

You cannot create an identifier that is the same as any reserved word. You can, however, declare an identifier that contains within it the letters of a reserved word (for example, the identifier DOT contains the reserved word DO).

Several categories of reserved words exist in Wang PC Pascal:

- Reserved words for standard-level Wang PC Pascal
- Reserved words added for extend-level Wang PC Pascal features
- Reserved words added for system-level Wang PC Pascal features
- Names of attributes
- Names of directives

See Appendix E for a complete list of reserved words.

11.4 UNUSED CHARACTERS

Wang PC Pascal does not use the following printing characters:

% & " ' ! ! <

You can, however, use them within comments or string literals.

Some nonprinting ASCII characters generate error messages if you use them in a source file other than in a comment or string literal. These are:

- The characters from CHR (0) to CHR (31), except the tab and form feed, CHR (9) and CHR (12), respectively
- The characters from CHR (127) to CHR (255)

The compiler treats the tab character, CHR (9), like a space and passes it on to the listing file. The compiler treats a form feed, CHR (12), like a space and starts a new page in the listing file.

CHAPTER 12 IDENTIFIERS

12.1 INTRODUCTION

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers. Constants, types, and variables can also have identifiers.

Some identifiers are predeclared; others you declare in a declaration section. Standard Pascal allows identifiers for the following elements of the Pascal language:

- Types
- Constants
- Variables
- Procedures
- Functions
- Programs
- Fields and tag fields in records

The following Wang PC Pascal features at the extend level also require identifiers:

- Super array types
- Modules
- Units
- Statement labels

An identifier consists of a sequence of alphanumeric characters or underscore characters. The first character must be alphabetic. Underscores are legal in identifiers, and are significant in all levels of Wang PC Pascal. Two underscores in a row or an underscore at the end of an identifier are also legal.

Identifiers

Subject to the restrictions noted below, identifiers can be as long as you want. They must, however, fit on a single line. On the Wang PC, the first 31 characters are significant. An identifier longer than 31 characters generates a warning message; the compiler ignores the excess characters.

Standard Pascal allows unsigned integers as statement labels. Statement labels have the same scope rules as identifiers (refer to Section 12.3). Leading zeros are not significant.

Extend-level Wang PC Pascal allows labels that are normal alphabetic identifiers.

The compiler passes the identifiers used for a program, module, or unit, as well as identifiers with the PUBLIC or EXTERN attribute, to the Linker. The operating system of a machine on which you plan to link and run a compiled Wang PC Pascal program may impose further identifier length restrictions on identifiers it uses as Linker global symbols. Furthermore, the object code listing and debugger symbol table may truncate variable and procedural identifiers to six characters.

Writing programs for use with other compilers and operating systems imposes an additional constraint on a program. Such a program must conform to the identifier restrictions for the worst possible case. For portability in general, the following practices are recommended:

- Make the first eight characters of all identifiers unique.
- Make the first six characters of external identifiers unique.
- Limit statement labels to four digits without leading zeros.

Identifiers of seven characters or fewer save space during compilation.

NOTE:

All identifiers the runtime system uses internally are four alphabetic characters followed by the characters QQ. Avoid this form when you create new names.

12.2 DECLARING AN IDENTIFIER

You declare identifiers and associate them with language objects in the declaration section of a program, module, interface, implementation, procedure, or function. Examples of identifiers, the objects they represent, and the syntax used to declare them are shown in Table 12-1. Although the details vary, the basic form of the declaration of the identifier for each of these elements is similar.

Identifiers

Table 12-1. Declaring Identifiers

Object	Identifier	Declaration
Program	Z	PROGRAM Z (INPUT, OUTPUT)
Module	XXX	MODULE XXX
Interface	UUU	INTERFACE; UNIT UUU
Implementation	UUU	IMPLEMENTATION of UUU
Constant	DAYS	CONST DAYS = 365
Type	LETTERS	TYPE LETTERS = 'A'..'Z'
Record fields	X, Y, Z	TYPE A = RECORD X, Y, Z : REAL END
Variable	J	VAR J : INTEGER
Label	1	LABEL 1
Label	HAWAII	LABEL HAWAII
Procedure	BANG	PROCEDURE BANG
Function	FOO	FUNCTION FOO: INTEGER

12.3 THE SCOPE OF IDENTIFIERS

Once you define an identifier, nothing can redefine it for the entire procedure, function, program, module, implementation, or interface in which you declare it. This holds true for any nested procedures or functions. An identifier's association must be unique within its scope; that is, it cannot name more than one thing at a time.

A nested procedure or function can redefine an identifier only if the procedure or function did not already use it. However, the compiler does not identify such redefinition as an error, but generally uses the first definition until the second occurs. A special exception for reference types is discussed in Section 17.1.5.

Identifiers

12.4 PREDECLARED IDENTIFIERS

A number of identifiers are already a part of the Wang PC Pascal language. This category includes the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures. You can use them freely without declaring them. They differ from reserved words, however, in that you can redefine them whenever you wish.

At the standard level, the following identifiers are predeclared:

ABS	FALSE	OUTPUT	ROUND
ARCTAN	FLOAT	PAGE	SIN
BOOLEAN	GET	PACK	SQR
CHAR	INPUT	PRED	SQRT
CHR	INTEGER	PUT	SUCC
COS	LN	READ	TEXT
DISPOSE	MAXINT	READLN	TRUE
EOF	NEW	REAL	TRUNC
EOLN	ODD	RESET	UNPACK
EXP	ORD	REWRITE	WRITE
WRITELN			

Features at the extend and system levels add the following to the list of predeclared identifiers in Wang PC Pascal:

1. String intrinsic procedures

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

2. Extend-level intrinsic procedures

ABORT	HIBYTE
BYWORD	LOBYTE
DECODE	LOWER
ENCODE	RESULT
EVAL	SIZEOF
UPPER	

3. System-level intrinsic procedures

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

Identifiers

4. Extend-level I/O

ASSIGN	READFN
CLOSE	READSET
DIRECT	SEEK
DISCARD	SEQUENTIAL
FCBFQQ	TERMINAL
FILEMODES	

5. INTEGER4 type

BYLONG	LOWORD
FLOATLONG	MAXINT4
HIWORD	ROUNDLONG
INTEGER4	TRUNCLONG

6. Super array type

LSTRING
NULL
STRING

7. WORD type

MAXWORD
WORD
WRD

8. Miscellaneous

ADRMEM	INTEGER2
ADSMEM	REAL4
BYTE	REAL8
INTEGER1	SINT

Identifiers

CHAPTER 13

INTRODUCTION TO DATA TYPES

13.1 UNDERSTANDING TYPES

A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly. For example, the types INTEGER and REAL are predeclared, while the type ARRAY [1..10] OF INTEGER is declared explicitly. An explicitly declared type may also have a type identifier; this requires a type declaration.

Types in Wang PC Pascal fall into four broad categories: simple, structured, reference, and procedural and functional types. Table 13-1 breaks down the types in each of these groups. The remainder of this chapter discusses types in general; Chapters 14 through 17 discuss the different groups in detail.

13.2 DECLARING DATA TYPES

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function).

A type declaration consists of an identifier followed by an equals sign and a type clause. The following are examples of type definitions:

```
TYPE LINE = STRING (80);
    PAGE = RECORD
        PAGENUM : 1..499;
        LINES : ARRAY [1..60] OF LINE;
        FACE : (LEFT, RIGHT);
        NEXTPAGE : ↑PAGE;
    END;
```

After declaring the data types, you declare variables of the types just defined in the VAR section of a program, procedure, function, module, or interface, or in the heading of a procedure or function. The following sample VAR section declares variables of the types in the preceding sample TYPE section:

```
VAR PARAGRAPH : LINE;
    BOOK : PAGE;
```

Table 13-1. Categories of Types in Wang PC Pascal

Category	Includes	Comments/Examples
Simple	Ordinal types INTEGER WORD CHAR BOOLEAN enumerated types subrange types REAL4, REAL8 INTEGER4	-MAXINT..MAXINT 0..MAXWORD CHR(0)..CHR(255) (FALSE,TRUE) e.g., (RED,BLUE) e.g., 100..5000 -MAXINT4..MAXINT4
Structured	ARRAY OF type general (OF any type) SUPER ARRAY (OF type) STRING (n) LSTRING (n) RECORD SET OF type FILE OF general (binary) files TEXT	[1..n] of CHAR [0..n] of CHAR Like FILE OF CHAR
Reference	Pointer Types ADR OF type ADS OF type	e.g., ^TREETIP Relative address Segmented address
Procedural and Functional		Only as parameter type

Because a type identifier is not defined until the compiler processes its declaration, a recursive type declaration such as the following is illegal:

```
T = ARRAY [0..9] OF T;
```

Reference types require a standard exception to this rule and are discussed in Chapter 17.

A special feature of Wang PC Pascal is a category called super types. A super type declaration determines the set of types that designators of that super type can assume; it also associates an identifier with the super type. Super type declarations also occur in the TYPE section. The only super types currently available in Wang PC Pascal are super arrays.

Introduction to Data Types

13.3 TYPE COMPATIBILITY

Wang PC Pascal follows the ISO standard for type compatibility, with some additional rules added for super array types, LSTRINGs, and constant coercions (forced changes in the type of a constant). Type transfer functions, to override the typing rules, are available with some Wang PC Pascal features.

Two types can be identical, compatible, or incompatible. An expression may or may not be assignment-compatible with a variable, value parameter, or array index.

13.3.1 Type Identity and Reference Parameters

Two types are identical if they have the same identifier or if a type definition like the following declares the identifiers equivalent:

```
TYPE T1 = T2;
```

Identical types are truly identical in Wang PC Pascal; there is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, and not on the way type identifiers declare or structure them. Thus, for example, T1 and T2 are not identical in the following declarations:

```
TYPE T1 = ARRAY [1..10] OF CHAR;
      T2 = ARRAY [1..10] OF CHAR;
```

Actual and formal reference parameters must be of identical types. If a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from it. Two record or array types must be identical for assignment.

The only exception to this rule is for strings. Here, actual parameters of type CHAR, STRING, STRING (n), LSTRING, and LSTRING (n) are compatible with a formal parameter of super array type STRING. Also, the type of a string constant changes to any LSTRING type with a large enough bound. For example, the type of 'ABC' changes to LSTRING (5) if necessary.

In addition, it is legal to pass an actual parameter of any FILE type to a formal parameter of a special record type FCBFQQ. Similarly, it is legal to pass an actual parameter of type FCBFQQ to a formal parameter of any file type. See Section 16.7 for a description of the FCBFQQ type.

STRING (n) is a shorthand notation for:

```
PACKED ARRAY [1..n] OF CHAR
```

The two types are identical. However, because variables with the type LSTRING receive special treatment in assignments, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [0..n] OF CHAR. The two types are not identical, compatible, or assignment-compatible. See Section 15.2.3 for further information on string types.

13.3.2 Type Compatibility and Expressions

Two simple or reference types are compatible if any of the following is true:

- They are identical.
- They are both ADR types.
- They are both ADS types.
- One is a subrange of the other.
- They are subranges of compatible types.

Two structured types are compatible if any of the following is true:

- They are identical.
- They are SET types with compatible base types.
- They are STRING derived types of equal length.
- They are LSTRING derived types.

However, two structured types are incompatible if any of the following is true:

- Either type is a FILE or contains a FILE.
- Either type is a super array type.
- One type is PACKED and the other is not.

Two values in an expression must be of compatible types. Most operators have additional limitations on the type of their operands; see Chapter 20 for details. A CASE index expression type must be compatible with all CASE constant values; two sets are never compatible if one is PACKED and the other is not.

13.3.3 Assignment Compatibility

Some types are implicitly compatible. This permits assignment across type boundaries. For instance, assume you declare the following variables:

```
VAR DESTINATION : T_DEST;
    SOURCE      : T_SOURCE;
```

SOURCE is assignment-compatible with DESTINATION (DESTINATION := SOURCE is permitted) if one of the following is true:

- T_SOURCE and T_DEST are identical types.
- T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.
- T_DEST is of type REAL and T_SOURCE is compatible with type INTEGER or INTEGER4.
- T_DEST is of type INTEGER4 and T_SOURCE is compatible with type INTEGER or WORD.

Also, if T_DEST and T_SOURCE are compatible structured types, then SOURCE is assignment-compatible with DESTINATION if one of the following is true:

- For SETs, every member of SOURCE is in the base type of T_DEST.
- For LSTRINGs, UPPER (DESTINATION) >= SOURCE.LEN.

Other than in the assignment statement itself, assignment compatibility is required in the following cases of implicit assignment:

- Passing value parameters
- READ and READLN procedures
- Control variable and limits in a FOR statement
- Super array type array bounds, and array indices

The compiler usually knows if assignment compatibility exists; an assignment generates simple instructions. However, the compiler cannot check some subrange, set, and LSTRING assignments until runtime, as the expression that contains these assignments does not have a value assignment until runtime. The range-checking switch checks assignment compatibility at runtime only if it is on.

CHAPTER 14

SIMPLE TYPES

The basic distinction between simple and structured data types is that you cannot divide simple types into two or more other types, while structured types (discussed in Chapter 15 and 16) consist of two or more other types. The simple data types fall into three categories: ordinal, REAL, and INTEGER4.

14.1 ORDINAL TYPES

Ordinal types are all finite. They include the following simple types:

- INTEGER
- WORD
- CHAR
- BOOLEAN
- enumerated types
- subrange types

INTEGER4, though it is finite, is not an ordinal type.

14.1.1 INTEGER

INTEGER values are a subset of the whole numbers that ranges from -MAXINT through 0 to MAXINT. MAXINT is the predeclared constant 32767 ($2^{15} - 1$) for the Wang PC. (The value -32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.) INTEGER is not a subrange of INTEGER4 (discussed in Section 14.3).

You must calculate expressions using a base type, not a subrange type. INTEGER type constants change to WORD type if necessary, but INTEGER variables do not. INTEGER values change to REAL or INTEGER4 in an expression, if necessary, but not to REAL. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

Simple Types

14.1.2 WORD

The WORD and INTEGER types are similar, differing chiefly in their range of values. Both are ordinal types. You can think of WORD values as either a group of 16 bits or as a subset of the whole numbers from 0 to MAXWORD (65535). The WORD type is a Wang PC Pascal feature that is useful in several ways, such as:

- To express values in the range from 32768 to 65535
- To operate on machine addresses
- To perform primitive machine operations, such as word ANDing and word shifting, without using the INTEGER type and exceeding the -32768 value

Unlike INTEGERS, all WORDs are nonnegative values. The WORD function changes any ordinal type value to WORD type. Like INTEGER values, WORD values in an expression can change to the INTEGER4 type, if necessary.

Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

1. As a signed value ranging from -32767 to +32767
2. As a positive value ranging from 0 to 65535

However, you must not mix WORD and INTEGER values in an expression (although doing so generates a warning message rather than an error message). WORD and INTEGER values are not assignment-compatible.

14.1.3 CHAR

In Wang PC Pascal, CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, Wang PC Pascal supports SET OF CHAR. Relational comparisons use the ASCII collating sequence.

Although the line-marker character in TEXT files is not part of the CHAR type in the ISO standard, Wang PC Pascal requires that you include a line-marker character (such as a carriage return).

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from 0 to 255. See Appendix D for a complete listing of the ASCII character set.

14.1.4 BOOLEAN

BOOLEAN is an ordinal type with only two (predeclared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is 0 and ORD (TRUE) is 1. This means that FALSE < TRUE.

You can redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former type in Boolean expressions and in IF, REPEAT, and WHILE statements.

No function exists for changing an ordinal value to a BOOLEAN value. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression:

```
ORD (value) <> 0
```

14.1.5 Enumerated Types

An enumerated type defines an ordered set of values. These values are constants; the identifiers that denote these values also enumerate them. Examples of enumerated type declarations are:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

At the extend-level, the READ and WRITE procedures and the ENCODE and DECODE functions operate on values of an enumerated type by treating the actual constant identifier as a string. This means that a program can read enumerated values directly.

The ORD function, at the standard level, can change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The RETYPE function, at system level, can change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN WRITELN ('TRUE BLUE')
```

The values the ORD function obtains from the constants of an enumerated type always begin with zero. Thus, the values for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED)   = 0
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

Simple Types

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations or commands. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers, since any arrays you index with the type or sets you base on the type change automatically. For example, interactive input of a command might occur when the program reads the enumerated type identifier that corresponds to a command. Because enumerated types are ordered, comparisons like RED < GREEN are possible. At times, access to the lowest and highest values of the enumerated type is useful with the LOWER and UPPER functions, as in the following example:

```
VAR TINT: COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
DO PAINT (TINT)
```

14.1.6 Subrange Types

A subrange type is a subset of an ordinal type. The type from which a subset derives is the "host" type; all subrange types are also ordinal types.

You can define a subrange type by giving the lower and upper bound of the subrange (in that order). The lower bound cannot be greater than the upper bound, but the bounds can be equal. The subrange type can be the index type of an array bound or the base type of a set (see Chapter 15). Examples of subranges along with their host ordinal type are:

<u>Subrange</u>	<u>Host Ordinal</u>
INTEGER	100..200
WORD	WRD(1)..9
CHAR	'A'..'Z'
enumerated type	RED..YELLOW

You can substitute a subrange clause for a list of values in the following circumstances:

- Set constants
- Set constructors
- CASE statement constants and record variant labels (at the extend level)

In addition to using the subrange type in array and set declarations, you can use it to help guarantee that the value of a variable is within acceptable bounds. If the range-checking switch is on during compilation, the compiler checks these bounds at runtime. For instance, if the logic of a program implies that a variable always has a value from 100 to 999, declaring it with a subrange ensures that the compiler never assigns the variable a value outside this range.

Simple Types

Declaring a subrange type may permit the compiler to allocate less room and use simpler operations. For example, declaring BOTTLES to be the INTEGER subrange 1..100 means that the compiler can allocate the type in 8 bits instead of 16.

Three subrange types are predeclared:

1. BYTE = WRD(0)..255; {8-bit WORD subrange}
2. SINT = -127..127; {8-bit INTEGER subrange}
3. INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types (see Section 17.1.2 for details) normally treat memory as an array of bytes. Since the BYTE type is really a subrange of the WORD type, however, the compiler calculates expressions with BYTE values using 16-bit instead of 8-bit arithmetic, if necessary.

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the math-checking switch is off), the compiler can optimize 16-bit arithmetic to 8-bit arithmetic. In general, using BYTE instead of WORD saves memory at the expense of BYTE-to-WORD conversions in expression calculations.

At the extend-level, subrange bounds can be constant expressions. Because the compiler assumes that the left parenthesis always starts an enumerated type declaration, the first expression in a subrange declaration must not start with a left parenthesis. For example:

```
TYPE {First two are permitted.}
    FEE = (A, B, C);
    FIE = M + 2 * N .. (P - 2) * N;
    {FOO is invalid as declared.}
    FOO = (M + 2) * N .. P - 2 * N;
```

14.2 REAL

REAL values are nonordinal values of a given range and precision: the range of allowable values depends on the target system. Wang PC Pascal implementations use a real number format with a 24-bit mantissa and an 8-bit exponent, giving about seven digits of precision and a maximum value of 1.701411E38. REAL constants must be between 1.0E-38 and 1.0E+38.

The current version of Wang PC Pascal includes expanded numeric data types for processing higher-precision real (and integer) numbers. For real numbers, this includes support for single- and double-precision real numbers according to the IEEE floating-point standard.

Simple Types

Standard Pascal provides a type REAL. Wang PC Pascal provides three real types: REAL, REAL4, and REAL8. The type REAL is always identical to either REAL4 or REAL8. The \$REAL:n metaccommand declares whether n is 4 or 8. {\$REAL:8} has the same effect as TYPE REAL = REAL8. The default type for REAL is normally REAL4, though you can change it (see Appendix A for details).

You can use any or all of these real number forms in a single program. However, programs that use REAL4 and REAL8, however, are not portable.

The REAL4 type is in 32-bit IEEE format, and the REAL8 type is in 64-bit IEEE format. The IEEE standard format is:

- REAL4 — Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa
- REAL8 — Sign bit, 11-bit binary exponent with bias of 1023, 52-bit mantissa

In both cases, the mantissa has a "hidden" most significant bit (always a one bit) and the mantissa represents a number greater than or equal to 1.0 but less than 2.0. An exponent of zero means a value of zero, and the maximum exponent means a value called NaN (not a number). Bytes are in "reverse" order; the byte with the lowest address byte is the least significant mantissa byte.

The REAL4 numeric range is seven significant digits (24 bits), with an exponent range of E-38 to E+38. The REAL8 numeric range includes over 15 significant digits (53 bits), with an exponent range of E-306 to E+306.

The exponent character can be D or d as well as E or e, so a number such as 12.34d56 is permitted. This minor extension provides compatibility with other Wang PC compiled languages. The D or d exponent character does not indicate double-precision (as it does in FORTRAN).

Wang PC Pascal converts REAL literals to REAL8 format and then to REAL4 as necessary (for example, for the compiler to pass as a CONST parameter or initialize a variable in a VALUE section). If you need actual REAL4 constants, you must declare them as REAL4 variables (by adding the READONLY attribute) and assign them a constant in a VALUE section.

The compiler passes both REAL4 and REAL8 values to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it evaluates the expression, assigns the result to a stack temporary, and passes the address of the temporary, which is usually more efficient than passing the value itself (especially in the REAL8 case).

Functions that return REAL values use the long return method; that is, the caller passes an additional, hidden, offset address of a stack temporary that will receive the result. This applies to all functions returning REAL4 or REAL8 values, both user-defined and intrinsic.

Simple Types

The Wang PC Pascal runtime library provides additional REAL functions to support Wang PC FORTRAN. These functions are available in Wang PC Pascal, but are not predeclared (see Chapter 23 for further information on these additional REAL functions and how to use them).

14.3 INTEGER4

INTEGER4 values, like INTEGER and WORD values, are a subset of the whole numbers. INTEGER4 values range from -MAXLONG to MAXLONG. MAXLONG is a predeclared constant with the value of 2,147,483,647 (that is, $2^{31} - 1$). The value -2,147,487,648 (that is, -2^{31}) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. Also, you cannot use INTEGER4 values to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, as is REAL. Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value that is out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must use the FLOATLONG function explicitly to make the conversion.

CHAPTER 15

ARRAYS, RECORDS, AND SETS

A structured type is composed of other types. The components of structured types are either simple types or other structured types. A structured type is characterized by the types of its components and by its structuring method. In Wang PC Pascal, a structured type can occupy up to 65534 bytes of memory.

The structured types in Wang PC Pascal are:

```

ARRAY <range> OF <type>
SUPER ARRAY <range> OF <type>
    STRING (n)
    LSTRING (n)
RECORD
SET OF <base-type>
FILE OF <type>

```

Because components of structures can be structured types themselves, you might have an array of arrays, a file of records containing sets, or a record containing a file and another record. This is an example of the data typing flexibility that provides Pascal with much of its linguistic power as a computing language.

The remainder of this chapter discusses arrays, records, and sets. See Chapter 16 for a discussion of files.

15.1 ARRAYS

An array type is a structure that consists of a fixed number of components. All of the components are of the same type (called the "component type").

Indices designate the elements of the array. Indices are values of the "index type" of the array. The index type must be an ordinal type: BOOLEAN, CHAR, INTEGER, WORD, subrange, or enumerated.

Arrays in Pascal are one-dimensional, but because the component type can also be an array, Wang PC Pascal supports n-dimensional arrays as well.

The following are examples of type declarations for arrays:

```
TYPE
  INT_ARRAY : ARRAY [1..10] OF INTEGER;
  ARRAY_2D  : ARRAY [0..7] OF ARRAY [0..8] OF 0..9;
  MORAL_RAY : ARRAY [PEOPLE] OF (GOOD, EVIL)
```

In the last declaration, PEOPLE is a subrange type, while GOOD and EVIL are enumerated constants.

A shorthand notation available for n-dimensional arrays makes the following statement the same as the second example in the preceding paragraph:

```
ARRAY_2D : ARRAY [0..7, 0..8] OF 0..9;
```

After declaring these arrays, you could assign the components of the arrays with statements such as these:

```
INT_ARRAY [10] := 1234;
ARRAY_2D [0,99] := 999;
MORAL_RAY [Machiavelli] := EVIL;
```

The whole of an n-dimensional PACKED array is packed; therefore, these statements are equivalent:

```
PACKED ARRAY [1..2, 3..4] OF REAL
```

```
PACKED ARRAY [1..2] OF PACKED ARRAY [3..4] OF REAL
```

See Chapter 17 for a discussion of packed types.

15.2 SUPER ARRAYS

A super array is an example of a Wang PC Pascal "super type." A super type is like a set of types or like a function that returns a type. Super types in general, and super arrays in particular, are features of Wang PC Pascal.

The super array type has several important uses. You can use them for any of the following purposes:

- To process strings. Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable-length strings. STRING handles fixed-length strings and strings that are more than 255 characters long.
- To allocate arrays of varying sizes dynamically. Otherwise, such arrays would need a maximum possible size allocation.
- As the formal parameter type in a procedure or function. Such a declaration allows you to use the procedure or function for a set or class of types, rather than for just a single fixed-length type.

Arrays, Records, and Sets

A super type identifier specifies the set of types the super type represents. A later type declaration may declare a normal type identifier as a type "derived" from that class of types. This derived type is like any other type.

A super array type declaration is an array type declaration prefixed with the keyword SUPER. Every array upper bound becomes an asterisk, as shown:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
```

After the preceding type declaration, you can declare the following variables:

```
VAR  ROW: VECTOR (10);
      COL: VECTOR (30);
      ROWP: ↑ VECTOR;
```

In this example, VECTOR is a super array type identifier. VECTOR (10) and VECTOR (30) are type designators that denote derived types. ROW and COL are variables of types derived from VECTOR. ROWP is a pointer to the super array type VECTOR.

Although the general concept of super types allows other "types of types," such as super subranges and super sets (in addition to super arrays), super types currently allow only an array type with parametric upper bounds. A super type is a class of types and not a specific type. Thus, in the VAR section of a program, procedure, or function, you cannot declare the variables to be of a super type; you must declare them as variables of a type derived from the super type. You can give a formal reference parameter in a procedure or function a super type, however; this allows the routine to operate on any of the possible derived types. (This kind of parameter is called a "conformant array" in other Pascals.)

You can also give a pointer referent type a super type. This allows a pointer to refer to any of the possible derived types. A pointer referent to a super type allows "dynamic arrays." The compiler allocates these arrays on the heap by passing their upper bound to the procedure NEW. See Chapter 17 for a discussion of pointer types and dynamic allocation. See Chapter 23 for a description of the procedure NEW.

The following is an example using the NEW procedure for dynamic allocation:

```
VAR STR_PNT: ↑SUPER PACKED ARRAY [1..*] OF CHAR;
    VEC_PNT: ↑SUPER ARRAY [0..*, 0..*] OF REAL;
.
NEW (STR_PNT, 12);
NEW (VEC_PNT, 9, 99);
```

An actual parameter in a procedure or function can be of a super type rather than a derived type, but only if the parameter is a reference parameter or pointer referent. (These are the only kinds of variables that can be of a super type rather than a derived type.) An example of super arrays follows:

Arrays, Records, and Sets

```

TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
{"VECTOR" is the super array type identifier.}

VAR X: VECTOR (12); Y: VECTOR (24); Z: VECTOR (36);
{X, Y, and Z are types derived from VECTOR.}

{Below, SUM accepts variables of all types}
{derived from the super type VECTOR.}
FUNCTION SUM (VAR V: VECTOR): REAL;
{V is the formal reference parameter of the}
{super type VECTOR.}

VAR S: REAL; I: INTEGER;
BEGIN
  S := 0;
  FOR I := 1 TO UPPER (V) DO S := S + V [I];
  SUM := S;
END;

BEGIN
  .
  .
  TOTAL := SUM (X) + SUM (Y) + SUM (Z);
  .
  .
END

```

The normal type rules for components of a super array type and for type designators that use a super array type allow the compiler to assign, compare, and pass components as parameters.

The UPPER function returns the actual upper bound of a super array parameter or referent. The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type the lower bound implies (such as MAXINT, MAXWORD). Two super array types are predeclared: STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

- LSTRING and STRING assignment
- LSTRING and STRING comparison
- LSTRING and STRING READs
- Access to the length of a STRING with the UPPER function
- Access to maximum length of an LSTRING with the UPPER function
- Access to LSTRING length with STR.LEN and STR[0]

Section 15.2.3 discusses these subjects.

Arrays, Records, and Sets

15.2.1 STRINGS

STRINGS are predeclared super arrays of characters:

TYPE STRING = SUPER PACKED ARRAY [1..*] OF CHAR;

A string literal such as 'abcdefg' automatically has the type STRING (n). The size of the array 'abcdefg' is 7; thus, the constant is of the STRING derived type, STRING (7).

Standard Pascal calls any packed array of characters with a lower bound of 1 a string and permits a few special operations on this type (such as comparison and writing) that you cannot do with other arrays.

In Wang PC Pascal, the super array notation STRING (n) is identical to PACKED ARRAY [1..n] OF CHAR (n can range from 1 to MAXINT). There is no default for n, because STRING means the super array type itself and not a string with a default length.

The identifier STRING is for a super array, so you can only use it as a formal reference parameter type or pointer referent type. The other super array restrictions apply; you cannot compare such a parameter or assign it as a whole.

The compiler can pass any variable (or constant) with the super array type STRING, or one of the types CHAR or STRING (n) or PACKED ARRAY [1..n] OF CHAR, to a formal reference parameter of super array type STRING. The compiler can also pass a variable of type LSTRING or LSTRING (n) to a formal reference parameter of type STRING. For a discussion of STRING as a formal reference parameter, see Section 15.2.3.

Standard Pascal supports the assignment, comparison, and writing of STRINGS. The extend-level permits reading STRINGS, including the super array type STRING and a derived type STRING (n). Reading a STRING inputs characters until the end of a line or STRING. If the end of the line occurs first, the compiler fills the rest of the STRING with blanks. Writing a string writes all of its characters.

The normal Pascal type compatibility rules are relaxed for STRINGS. The compiler can compare or assign any two variables or constants with the type PACKED ARRAY [1..n] OF CHAR or the type STRING (n) if the lengths are equal. Because the length of a STRING super array type varies, however, comparisons and assignments are illegal.

The following is an illegal STRING assignment:

```
PROCEDURE CANNOT_DO (VAR S : STRING);
VAR STR : STRING (10);
BEGIN
  STR := S
  {This assignment is illegal because}
  {the length of S may vary.}
END;
```

Arrays, Records, and Sets

The PACKED prefix in the declaration PACKED ARRAY [1..n] OF CHAR, as defined in the ISO standard, normally implies that the compiler cannot pass a component as a reference parameter. In Wang PC Pascal, this restriction does not apply.

To keep conformance to the ISO standard, passing the CHAR component of a STRING as a reference parameter is an "error not caught." The index type of a string is officially INTEGER, but WORD type values can also index a STRING. Many string-processing applications take advantage of the LSTRING type, described in Section 15.2.2.

Chapter 23 discusses a number of intrinsic procedures and functions for strings. Many of the procedures and functions described work on STRINGS; some apply only to LSTRINGS.

15.2.2 LSTRINGS

The LSTRING feature in Wang PC Pascal allows variable-length strings. LSTRING (n) is predeclared as:

TYPE LSTRING = SUPER PACKED ARRAY [0..*] OF CHAR

A variable with the explicit type PACKED ARRAY [0..n] OF CHAR, however, is not "identical" to the type LSTRING (n), even though they are structurally the same. There is no default for n; the range of n is from 0 to 255. The usual array notation can access characters in an LSTRING.

Internally, LSTRINGS contain a length (L), followed by a string of characters. Element zero of the LSTRING contains the length, which can vary from 0 to the upper bound. The length of an LSTRING variable T is accessible as T[0] with type CHAR, or as T.LEN with type BYTE. String constants of type CHAR or STRING (n) change automatically to type LSTRING.

The predeclared constant NULL is the empty string, LSTRING (0). NULL is the only constant with type LSTRING; there is no way to define other LSTRING constants. As with STRINGS, the compiler can pass a CHAR component of an LSTRING as a reference parameter, and WORD and INTEGER values can index an LSTRING.

Several operations work differently on LSTRINGS than on STRINGS. You can assign an LSTRING to any other LSTRING if the current length of the right side is not greater than the maximum length of the left side. Similarly, you can pass an LSTRING as a value parameter to a procedure or function if the current length of the actual parameter is not greater than the maximum length the formal parameter specifies. If the range-checking switch is on, the compiler checks the assignment of LSTRINGS and the passing of LSTRING (n) parameters. The actual number of bytes the compiler assigns or passes is the minimum of the upper bounds of the LSTRINGS.

Neither side in an LSTRING assignment can be a parameter of the super array type LSTRING; both must be types derived from it.

Arrays, Records, and Sets

The following are examples of LSTRING assignments:

```

{Declaring the variables}
VAR A : LSTRING (19);
    B : LSTRING (14);
    C : LSTRING (6);
.
{Assigning the variables}
A := '19 character string';
B := '14 characters';
C := 'shorty';
A := B;
{This is legal, because the length of B}
{is less than the maximum length of A.}
C := A;
{This is illegal, because the length of A}
{is greater than the maximum length of C.}

```

You can compare any two LSTRINGs, including super array type LSTRINGs (the only super array type comparison allowed). Reading an LSTRING variable inputs characters until the end of the current line or the end of the LSTRING, and sets the length to the number of characters the LSTRING variable read. Writing from an LSTRING writes the current length string.

15.2.3 Using STRINGS and LSTRINGs

This section describes the STRING and LSTRING operations directly supported by the compiler. An annotated program at the end of this section illustrates the use of STRINGS and LSTRINGs in context. Chapter 23 provides descriptions of the following string procedures and functions:

CONCAT	INSERT
COPYLST	POSITN
COPYSTR	SCANEQ
DELETE	SCANNE

At the system level of Wang PC Pascal, the procedures FILLC, FILLSC, MOVCL, MOVESL, MOVER, and MOVESR also operate on strings.

Wang PC Pascal supports STRINGS and LSTRINGs directly in the following ways:

- **Assignment** — You can assign any LSTRING value to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, the compiler assigns only the target length, and a runtime error occurs if the range-checking switch is on. You can assign a STRING value to a STRING variable, as long as the length of both sides is the same and neither side is the super array type STRING. Passing either STRING or LSTRING as a value parameter is much like making an assignment.
- **Comparison** — The LSTRING operators (<, <=, >, >=, and <>,) use the length byte for string comparisons; the operands can be of different lengths. Two strings must be the same length to be equal. If two strings of different lengths are equal up to the length of the shorter one, the shorter is less than the longer one. The operands can be of the super array type LSTRING. For STRINGS, the same relational operators are available, but the lengths must be the same and operands of the super array type STRINGS are illegal.
- **READs and WRITEs** — READ LSTRING reads until the LSTRING is full or until the end-of-line. The current length becomes the number of characters READ LSTRING read. WRITE LSTRING uses the current length. See also READSET (described in Chapter 24), which reads into an LSTRING as long as input characters are in a SET OF CHAR. READ STRING pads with spaces if the line is shorter than the STRING. WRITE STRING writes all the characters in the string. Both READ and WRITE allow the super array types STRING and LSTRING, as well as their derived types.
- **Length access** — You can access the current length of an LSTRING variable T with T.LEN, which is of type BYTE, or with T[0], which is of type CHAR. This notation can assign a new length, as well as determine the current length. The UPPER function finds the maximum length of an LSTRING or the length of a STRING. This is especially useful for finding the upper bound of a super array reference parameter or pointer referent.

You cannot assign or compare mixed STRINGS and LSTRINGs unless the STRING is constant. You can assign STRINGS to LSTRINGs, or vice versa, with one of the move routines or with the COPYSTR and COPYLST procedures. Since constants of type STRING or CHAR change automatically to type LSTRING if necessary, LSTRING constants are normal STRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

In the sample program at the end of this section, all STRING parameters (CONST or VAR) can take either a STRING or an LSTRING; all LSTRING parameters are VAR LSTRING and must take an LSTRING variable.

Arrays, Records, and Sets

A "special transformation" lets you pass an actual LSTRING parameter to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING. Therefore, if LSTR (in the following example) is of type LSTRING (n) or LSTRING, a formal reference parameter of type STRING can pass it to a procedure or function.

```
VAR LSTR : LSTRING (10);
.
PROCEDURE TIE_STRING (VAR STR : STRING);
.
TIE_STRING (LSTR);
```

In this case, UPPER (STR) is equivalent to LSTR.LEN.

Procedures and functions with reference parameters of super type STRING can operate equally well on STRINGS and LSTRINGs. The only reason to declare a parameter of type LSTRING is when you must change the length. Normally, an LSTRING is either a VAR or a VARS parameter in a procedure or function, since a CONST or CONSTS parameter of type LSTRING cannot change.

An example of a program that uses STRINGS and LSTRINGs follows:

```
PROGRAM STRING_SAMPLE;

PROCEDURE STRING_PROC (CONST S: STRING); BEGIN END;
PROCEDURE LSTRING_PROC (CONST S: LSTRING); BEGIN END;

VAR
  CHR1VAR: CHAR;
  STR5VAR: STRING (5);
  LST5VAR: LSTRING (5);
  LST9VAR: LSTRING (9);
  STR4VAR: PACKED ARRAY [1..4] OF CHAR;
  STR6VAR: PACKED ARRAY [1..6] OF CHAR;

BEGIN

  {Look at all the kinds of strings a}
  {CONST STRING parameter takes.}
  STRING_PROC ('A');
  {Character constant is OK.}
  STRING_PROC (CHR1VAR);
  {Character variable is OK.}
  STRING_PROC ('STRING');
  {STRING constant is OK.}
  STRING_PROC (STR5VAR);
  {STRING variable is OK.}
  STRING_PROC (LST5VAR);
  {LSTRING variable is OK.}
```

Arrays, Records, and Sets

```

{However, a CONST LSTRING parameter cannot take}
{non-LSTRING variables.}
LSTRING_PROC ('A');
{Character constant is OK.}
LSTRING_PROC (CHR1VAR);
{Character variable is not OK!}
LSTRING_PROC ('STRING');
{STRING constant is OK.}
LSTRING_PROC (STR5VAR);
{STRING variable is not OK!}
LSTRING_PROC (LST5VAR);
{LSTRING variable is OK.}

```

```

{Assignments to a STRING variable are limited}
{to the same type.}
STR5VAR := 'A';
{Character constant is not OK!}
STR5VAR := CHR1VAR;
{Character variable is not OK!}
STR5VAR := 'TINY';
{STRING constant too small.}
STR5VAR := 'RIGHT';
{Both sides have five characters; OK.}
STR5VAR := 'longer';
{Not OK; STRING constant is too large.}
STR5VAR := LST5VAR;
{Not OK; you cannot assign LSTRINGs to STRINGs.}
COPYSTR (LST5VAR, STR5VAR);
{COPYSTR is an intrinsic procedure.}
STR5VAR := STR4VAR;
{Not OK; STRING variable is too small.}
COPYSTR (STR4VAR, STR5VAR);
{COPYSTR is OK; padding of space in STR5VAR[5].}
STR5VAR := STR5VAR;
{OK; both sides have five characters.}
STR5VAR := STR6VAR;
{Not OK; STRING variable is too large.}

```

```

{Assignments to an LSTRING variable, however,}
{are more flexible.}
LST5VAR := 'A';
{Character constant is OK.}
LST5VAR := CHR1VAR;
{Character variable is not OK!}

```

Arrays, Records, and Sets

```

LST5VAR := 'TINY';
{Smaller STRING constant is OK.}
LST5VAR := 'RIGHT';
{Same length STRING constant is OK.}
LST5VAR := 'LONGER';
{This gives an error at runtime only; OK for now.}
LST5VAR := LST9VAR;
{This may give an error at runtime; OK for now.}
LST9VAR := LST5VAR;
{This isn't even checked at runtime; always OK.}
LST5VAR := STR5VAR;
{Not OK; you cannot assign a STRING variable to an
LSTRING variable.}
COPYLST (STR5VAR, LST5VAR);
{This is the way to copy a STRING variable
to an LSTRING.}

```

END.

15.3 RECORDS

A record structure acts as a template for conceptually related data of different types. The record type itself is a structure that consists of a fixed number of components, usually of different types.

Each component of a record type is a field. The definition of a record type specifies the type and an identifier for each field within the record. Because the scope of these field identifiers is the record definition itself, they must be unique within the declaration. The field values associated with field identifiers are accessible with record notation or with the WITH statement. For example, you could declare the following record type:

```

TYPE LP = RECORD
    TITLE : LSTRING (100);
    ARTIST : LSTRING (100);
    PLASTIC : ARRAY
        [1..SONG_NUMBER] OF SONG_TITLE
    END

```

You could then declare a variable of the type LP, as follows:

```
VAR BEATLES_1 : LP;
```

Finally, you could access a component of the record with either field notation or the WITH statement (note the period that separates field identifiers):

```

BEATLES_1.TITLE := 'Meet The Beatles';
WITH BEATLES_1 DO
    PLASTIC[1] := 'I Wanna Hold Your Hand'

```

Arrays, Records, and Sets

15.3.1 Variant Records

A record can have several variants. In this case, a field called the tag field indicates which variant to use. The tag field may or may not have an identifier and storage in the record. Some operations, such as the NEW and DISPOSE procedures and the SIZEOF function, can specify a tag value even if the tag is not part of the record.

The following are examples of variant records:

```

TYPE OBJECT = RECORD
  X, Y: REAL;
  CASE S: SHAPE OF
    SQUARE: (SIZE, ANGLE: REAL);
    CIRCLE: (DIAMETER: REAL)
  END;

FOO = RECORD
  CASE BOOLEAN OF
    TRUE: (I, J: INTEGER);
    FALSE: (CASE COLOR OF
      BLUE: (X: REAL);
      RED: (Y: INTEGER4));
  END;

```

Each record can have only one variant part, which must be the last field of the record. This variant part, however, can also have a variant, and so on, to any level. All field identifiers in a given record type must be unique, even in different variants. For example, after declaring the record types above, you could create and then assign to the variables shown in the following example:

```

VAR O, P : OBJECT;
    F, G : FOO;

BEGIN
  O.DIAMETER := 12.34; {CASE of CIRCLE}
  P.SIZE := 1.2;       {CASE of SQUARE}
  F.I := 1; F.J := 2;  {CASE of TRUE}
  G.X := 123.45;       {CASE of FALSE and BLUE}
  G.Y := 678999        {CASE of FALSE and RED;}
                      {this overwrites G.X.}

END;

```

The most recent ISO standard requires every possible tag field value to select some variant. Therefore, it is illegal to include CASE INTEGER OF and omit a variant for every possible INTEGER value. Such an omission does not generate an error in Wang PC Pascal, however.

Arrays, Records, and Sets

Wang PC Pascal supports full CASE constant options in the variant clause; that is, a list of constants can define a case. At the extend-level, subranges and the OTHERWISE statement can also define a case. OTHERWISE applies to the last variant in the list and is not followed by a colon. You can also declare an empty variant, such as POINT:() or OTHERWISE (). You can even declare an entirely empty record type, although the compiler issues a warning whenever it uses the record.

The ISO standard defines a number of errors that relate to variant records; Wang PC Pascal may not detect these errors, even if the tag-checking switch is on. (The tag-checking switch generates code for each variant field to check that the tag value is correct.) In the record type declaration of OBJECT in the previous example, any use of SIZE generates a check that S = SQUARE. However, in the case of FOO, the compiler cannot check uses of I because Wang PC Pascal does not allocate the BOOLEAN tag field.

The ISO standard further declares that when a "change of variant" occurs (such as when the compiler assigns a new tag value), all the variant fields become undefined. However, Wang PC Pascal does not set the fields to an uninitialized value when the compiler assigns a new tag. Therefore, using a variant field with an undefined value does not generate an error in Wang PC Pascal.

Wang PC Pascal also does not enforce various restrictions on record variable allocation on the heap with the long form of the NEW procedure (see Chapter 23 for details). Wang PC Pascal does, however, check an assignment to such a "short record" to see that only the short record itself is modified in the heap.

It is legal in Wang PC Pascal to release a record allocated with the long form of NEW by using the short form of DISPOSE. This generates an error in ISO Pascal. It is also legal in Wang PC Pascal to DISPOSE of a record an active WITH statement either passes as a reference parameter or uses.

Wang PC Pascal generates a warning message if you attempt either of these operations:

1. Declaring a variant that contains a file is not safe; any change to the file's data using a field in another variant may lead to I/O errors, even if you have closed the file. In the following example, any use of R leads to errors in F:

```

RECORD CASE BOOLEAN OF
  TRUE : (F: FILE OF REAL);
  FALSE : (R:ARRAY [1..100] OF REAL);
END;
```

Arrays, Records, and Sets

2. Giving initial data to several overlapping variants in a variable in a VALUE section could have unpredictable results. In the following example, the initial value of LAP is uncertain:

```
VAR LAP : RECORD CASE BOOLEAN OF
  TRUE : (I: INTEGER4);
  FALSE : (R: REAL);
END;
VALUE LAP.I := 10; LAP.R := 1.5;
```

15.3.2 Explicit Field Offsets

Wang PC Pascal lets you assign explicit byte offsets to the fields in a record. This system-level feature can be useful for accessing routines in other languages, since control block formats may not conform to the usual Wang PC Pascal field allocation method. However, because it also permits unsafe operations such as overlapping fields and word values at odd byte boundaries, you should not use this feature unless the interface is necessary.

The following is an example of assignment of explicit byte offsets:

```
TYPE CPM = RECORD
  NDRIVE [00]: BYTE;
  FILENM [01]: STRING (8);
  FILEXT [09]: STRING (3);
  EXTENT [12]: BYTE;
  CPMRES [13]: STRING (20);
  RECNUM [33]: WORD;
  RECOVF [35]: BYTE;
END;

OVERLAP = RECORD
  BYTEAR [00]: ARRAY [0..7] OF BYTE;
  WORDAR [00]: ARRAY [0..3] OF WORD;
  BITSAR [00]: SET OF 0..63;
END;
```

In the above example, the offset appears in brackets; this is similar to attribute notation. The number is the byte offset to the start of the field. Some target machines may not allow access to a 16-bit value at an odd address, but this does not generate an error in Wang PC Pascal.

If you assign an offset to one field, you must assign offsets to all fields. The compiler picks an arbitrary value for any offset that you omit. Although the compiler processes a declaration that includes both offsets and variant fields, you should use only one or the other in a given program.

Arrays, Records, and Sets

Although you can control field overlap completely with explicit offsets, variants provide the long forms of the procedures NEW, DISPOSE, and SIZEOF. If you want to allocate records of different length, use the RETYPE and GETHQQ procedures instead of variants and the long form of NEW. For example:

```
CPMPV := RETYPE (CPMP, GETHQQ (36));
```

The compiler supports structured constants for record types with explicit offsets. Internally, the compiler rounds odd length fields greater than one to the next even length. For example:

```
ODDR = RECORD
    F1[00] : STRING (3);
    F2[03] : CHAR
END;
```

In this example, field F1 is 4 bytes long, so an assignment to F1 overwrites F2. In such a record, the compiler assigns all odd length fields first.

15.4 SETS

A set type defines the range of values that a set can assume. This range of assumable values is the "power set" of the base type you specify in the type definition. The power set is the set of all possible sets that you can compose from an ordinal base type. The null set, [], is a member of every set.

If you declare the following set types:

```
TYPE HUES = SET OF COLOR;
    CAPS = SET OF 'A'..'Z';
    MATTER = SET OF (ANIMAL, VEGETABLE, MINERAL);
```

Then you declare variables like the following:

```
VAR FLAG : HUES;
    VOWELS : CAPS;
    LIVE : MATTER;
```

Finally, you could assign these set variables:

```
FLAG := [RED, WHITE, BLUE];
VOWELS := ['A', 'E', 'I', 'O', 'U'];
LIVE := [ANIMAL, VEGETABLE];
```

You must enclose the set elements in brackets. This practice differs from using parentheses to enclose the base enumerated type in a set type declaration.

Generated in-line code or routines in the set unit implement set operations directly. See Chapter 20 for a complete discussion of operations on sets.

The ORD value of the base type can range from 0 to 255. Thus, SET OF CHAR is legal, but SET OF 1942..1984 is not.

Sets whose maximum ORD value is 15 (that is, sets that fit into a WORD) are usually more efficient than larger ones. Also, if the range-checking switch is on, passing a set as a value parameter invokes a runtime compatibility check unless the formal and actual sets have the same type.

Sets provide a clear and efficient way of giving several qualities or attributes to an object. In another programming language, you could assign each quality a power of two:

```
READY = 1
GETSET = 2
ACTIVE = 4
DONE = 8
```

You could then assign the qualities with a statement such as:

```
X := READY + ACTIVE
```

and then test them using OR and AND as bitwise operators with a statement such as:

```
IF ((X AND ACTIVE) <> 0) THEN WRITELN ('GO FISH')
```

The equivalent declaration in Wang PC Pascal is:

```
QUALITIES = SET OF (READY, GETSET, ACTIVE, DONE);
```

You can then assign the qualities with `X := [GETSET, ACTIVE]` and test them with the following operations:

```
IN      tests a bit
+       sets a bit
-       clears a bit
```

For example, one appropriate construction is:

```
IF ACTIVE IN X THEN WRITELN ('GO FISH')
```

You can also use SET OF 0..15 to test and set the bits in a WORD. Using WORDs both as a set of bits and as the WORD type requires giving two types to the word with a variant record, the RETYPE function, or an address type.

The compiler assigns the bits in a set starting with the most significant bit in the byte with the lowest address. Thus, the set [0, 7, 8, 15] has the WORD value `#80 + #01 + #8000 + #0100`.

Arrays, Records, and Sets

CHAPTER 16

FILES

A file is a structure that consists of a sequence of components that are all of the same type. It is through files that Wang PC Pascal communicates with a given operating system. Therefore, you must understand the FILE type in order to perform input to and output from a program.

16.1 DECLARING FILES

As with any other type, you must declare a file variable in order to use it. Declaring a FILE type, however, does not fix the number of components in a file. Examples of FILE declarations are:

```
TYPE F1 = FILE OF COLOR;  
      F2 = FILE OF CHAR;  
      F3 = TEXT;
```

Conceptually, a file is simply another data type, like an array, but with no bounds and with only one component accessible at a time. However, a file usually corresponds to one of the following:

- Disk files
- Terminals
- Printers
- Other input and output devices

This implies the following restriction in Pascal: a FILE OF FILE is illegal, directly or indirectly. Other structures, such as a FILE OF ARRAYS or an ARRAY OF FILES, are legal.

Most Pascal implementations connect file variables to the data files of the operating system. Wang PC Pascal always uses the target operating system to access files, but does not impose additional formatting or structure on operating system files.

Wang PC Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer referents (allocated on the heap). Except for files in super arrays, the compiler generates code that initializes a file upon allocation and closes a file when allocation ceases. This initialization call occurs automatically in most cases. A file in a module or uninitialized unit's interface, however, gets its initialization call only if you call the module or unit identifier as a procedure. File declarations in such cases cause the following compiler warning: "Contains file initialize module". Only a file in an interface of an uninitialized unit does not generate this warning.

Wang PC Pascal sets up the standard files, INPUT and OUTPUT (refer to Section 16.5). In standard Pascal, you must specify files in the program header; when you run your program, the runtime system prompts you for file names. At the extend-level, you can use the ASSIGN and READFN procedures to give explicit operating system file names to files not in the program header.

Do not use files in record variants or super array types; the compiler issues a warning if you do. A value cannot assign, compare, or pass a file variable unless you declare and pass it as a reference parameter.

At the extend-level, you can indicate a file's access method or other characteristics by specifying the mode of the file. The mode is a value of the predeclared enumerated type FILEMODES. The modes available normally include the three base modes: SEQUENTIAL, TERMINAL, and DIRECT. All files, except INPUT and OUTPUT are SEQUENTIAL mode by default. INPUT and OUTPUT are TERMINAL mode by default.

16.2 THE BUFFER VARIABLE

Every file F has an associated buffer variable F[↑]. Figure 16-1 represents a buffer variable and its associated file.

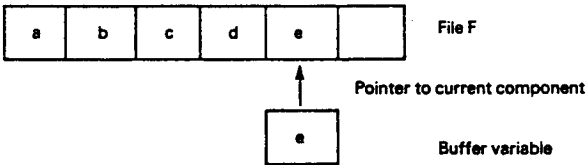


Figure 16-1. Buffer Variable and File

The procedures GET and PUT use this buffer variable to read from and write to files. GET copies the current component of the file to the buffer variable. PUT does the opposite; that is, PUT copies the value of the buffer variable to the current component.

You can reference a buffer variable (that is, fetch or store its value) like any other Wang PC Pascal variable. This allows execution of assignments like the following:

```
FT := 'z'
C := FT
```

You can pass a file buffer variable as a reference parameter to a procedure or function, or use it as a record in a WITH statement. The compiler may not update the file buffer variable correctly if the file position changes within the procedure, function, or WITH statement. The compiler issues a warning message to alert you to this possibility. For example, the following use of a file buffer variable generates a warning during compilation:

```
VAR A : TEXT;
PROCEDURE CHAR_PROC (VAR X : CHAR);
.
CHARPROC (A↑);
{Warning issued here}
```

Two special internal mechanisms exist in Wang PC Pascal, lazy evaluation and concurrent I/O. Lazy evaluation allows interactive terminal input in a natural way; concurrent I/O allows overlapping of I/O along with program execution. Lazy evaluation is applied to all ASCII structured files and is necessary for natural terminal input. Concurrent I/O is applied to all binary structured files and is necessary for some operating systems that support overlapping input and output. Both mechanisms generate a runtime call that executes before any use of the buffer variable. See Sections 24.1.5 and 24.1.6 for complete details.

16.3 FILE STRUCTURES

Wang PC Pascal files have two basic structures: binary and ASCII. These two structures correspond to raw data files and text files, respectively.

16.3.1 Binary Structure Files

The Pascal data type FILE OF <type> corresponds to Wang PC Pascal binary structure files. These, in turn, correspond to unformatted operating system files.

Under operating systems that divide files into records, every record is one component of the file type (not to be confused with the record type). Primitive procedures such as GET and PUT operate on a record basis. Under operating systems without their own record structure, the primitive procedures GET and PUT transfer a fixed number of bytes per call, equal to the length of one component. See Section 16.4 for further discussion of binary files.

16.3.2 ASCII Structure Files

The Pascal data type TEXT corresponds to Wang PC Pascal ASCII structure files. These in turn correspond to textual operating system files (called text files in this manual).

The Pascal TEXT type is like a FILE OF CHAR, except that groups of characters are organized into lines and, to a lesser extent, pages. Primitive file procedures, such as GET and PUT, always operate on a character basis.

Under operating systems that divide files into records, however, every record is a line (not a character). Even in operating systems that do not have their own record structure, other languages and utilities have some way of organizing bytes into lines of characters.

Wang PC Pascal provides a number of special functions and procedures that use this line-division feature. Because Wang PC Pascal does not impose any additional formatting on operating system files of most modes (including SEQUENTIAL, TERMINAL, and DIRECT), programs in other languages can generate and use these files.

A line marker divides Pascal text files (files of type TEXT) into lines. This line marker is conceptually a character not of the type CHAR. In theory, a text file can contain any value of type CHAR. Under some operating systems, however, writing a particular character (such as CHR (13), carriage return, or CHR (10), line feed) may terminate the current line (record). This character value is the line marker in this case and always looks like a blank.

At the extend-level, a declaration for a text file can include an optional line length. Setting the line length, which sets record length, is necessary for DIRECT mode text files only. You can specify line length for other modes, but doing so has no effect.

Specify the line length of a text file as a constant in parentheses after the word TEXT:

```
TYPE NAMEADDR = TEXT (128);
    DEFAULTTX = TEXT;
    SMALLBUF = TEXT (2);
```

16.4 FILE ACCESS MODES

The file access modes in Wang PC Pascal are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode files are available at the standard level; all three, including DIRECT mode, are available at the extend-level. SEQUENTIAL and TERMINAL mode ASCII structure files can have variable-length records (lines); DIRECT mode files must have fixed-length records or lines.

The declaration of a file in Pascal implies its structure, but not its mode. For example, FILE OF STRING (80) indicates binary structure; TEXT indicates ASCII structure. An assignment such as F.MODE := DIRECT sets the mode; this only works at the extend-level and is only needed to set DIRECT mode.

16.4.1 TERMINAL Mode Files

TERMINAL mode files always correspond to an interactive terminal or printer. The compiler opens TERMINAL mode files at the beginning of the file for either reading or writing. The compiler accesses records one after the other until it reaches the end of the file.

Operation of TERMINAL mode input for terminals depends on the file structure (ASCII or binary). For ASCII structure (type TEXT), the compiler reads entire lines at one time. This permits the usual operating system intraline editing, including backspace, advance cursor, and cancel. Characters appear on the screen (echoing) as you type the line.

Because the compiler reads an entire line at once, however, you cannot read the characters as you type them, invoke several prompts and responses on the same line, and so on.

For binary structure TERMINAL mode (usually type FILE OF CHAR), you can read characters as you type them. No intraline editing or echoing is possible. This method permits screen editing, menu selection, and other interactive programming on a keystroke, rather than line, basis.

TERMINAL mode files use lazy evaluation to manage normal interactive reading of the terminal keyboard. See Section 24.1.5 for details.

16.4.2 SEQUENTIAL Mode Files

SEQUENTIAL mode files are generally disk files or other sequential access devices. As with TERMINAL mode files, the compiler opens SEQUENTIAL mode files at the beginning of the file for either reading or writing, and accesses records one after another until the end of the file. Standard Pascal files are in SEQUENTIAL mode by default (except for INPUT and OUTPUT).

16.4.3 DIRECT Mode Files

DIRECT mode files are generally disk files or other random access devices. DIRECT mode files and the ability to access the mode of a file are available at the extend-level of Wang PC Pascal.

DIRECT mode ASCII structure files, as well as all binary structure files, have fixed-length records, where a record is either a line or file component. (The term record here refers not to the normal Pascal record type, but to a disk structuring unit.) DIRECT files are always open for both reading and writing, and the compiler can access records randomly by record number. Records begin with record number one; there is no record number zero.

16.5 THE PREDECLARED FILES INPUT AND OUTPUT

Two files, INPUT and OUTPUT, are predeclared in every Wang PC Pascal program. These files receive special treatment as program parameters and must appear as parameters in the program heading; for example:

```
PROGRAM ACTION (INPUT, OUTPUT);
```

If no program parameters exist and the program does not use the files INPUT and OUTPUT, the heading can look like this:

```
PROGRAM ACTION;
```

You should include INPUT and OUTPUT as program parameters if you use them, either explicitly or implicitly, in the program itself; for example:

```
WRITE (OUTPUT, 'Prompt: ') {Explicit use}
WRITE ('Prompt: ')         {Implicit use}
```

These examples generate a warning if you do not declare OUTPUT in the program heading. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

Although you can redefine the identifiers INPUT and OUTPUT, the file that text file input and output procedures and functions assume (for example, READ, EOLN) is the predeclared definition. This generates the procedures RESET (INPUT) and REWRITE (OUTPUT) automatically, whether INPUT and OUTPUT are present as program parameters or not (you can also use these procedures explicitly).

The INPUT and OUTPUT files have ASCII structure and TERMINAL mode. They are initially connected to your terminal and opened automatically. At the extend-level of Wang PC Pascal, you can change these characteristics if you wish.

16.6 EXTEND-LEVEL I/O

A file variable in Wang PC Pascal is really a record of type FCBFQQ, called a file control block. At the extend-level, a few standard fields in this record help you manage file modes and error trapping. The system level allows additional fields and the record type FCBFQQ itself, as described in Section 16.7. Along with access to certain FCB fields, extend-level I/O also includes the following procedures:

ASSIGN	READFN
CLOSE	READSET
DISCARD	SEEK

See Section 24.3 for a description of these procedures.

Use the normal record field syntax to access FCB fields. For a file F, these fields are F.MODE, F.TRAP, and F.ERRS. You can change or examine these fields at any time. The following paragraphs discuss these fields.

- **F.MODE: FILEMODES** -- This field contains the mode of the file: SEQUENTIAL, TERMINAL, or DIRECT. These values are constants of the predeclared enumerated type FILEMODES. The file system uses the MODE field only during RESET and REWRITE. Thus, changing the MODE field of an open file has no effect. Except for INPUT and OUTPUT, which have TERMINAL mode, a file's mode is SEQUENTIAL by default.

RESET and REWRITE change the mode from SEQUENTIAL to TERMINAL if they discover that the device you are opening is a terminal or printer and if the target operating system allows it. This is useful in programs designed to work either interactively or in batch mode. You must set DIRECT mode before RESET or REWRITE if you plan to use SEEK on a file.

- **F.TRAP: BOOLEAN** -- If this field is TRUE, error trapping for file F starts. Then, if an input/output error occurs, the program does not abort and you can examine the error code. Initially, F.TRAP is FALSE. If F.TRAP is FALSE and an I/O error occurs, the program aborts.
- **F.ERRS: WRD(0)..15** -- This field contains the error code for file F. An error code of zero means no error; values from 1 to 15 imply an error condition. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the program immediately aborts if F.TRAP is FALSE. If F.TRAP is TRUE, however, the compiler ignores the attempted file operation and the program continues.

CLOSE and DISCARD do not examine the initial value of F.ERRS, so they do not cause an immediate abort. Nevertheless, if CLOSE or DISCARD themselves generate an error condition, F.TRAP determines whether to trap the error or to abort.

An operation the compiler ignores because of an error condition does not change the file itself, but may change the buffer variable or READ procedure input variables. See Appendix H for a complete listing of error messages and warnings.

You can also set the line length for a text file at the extend-level, as shown:

```
TYPE SMALLBUF = TEXT (16);
VAR RANDOMTEXT: TEXT (132);
```

Declaring line length applies only to DIRECT mode ASCII structure files, where the line length is the record length in reading and writing. Setting the line length has no effect on other ASCII files.

16.7 SYSTEM-LEVEL I/O

At the system level of Wang PC Pascal, you can call procedures and functions that have a formal reference parameter of type FCBFQQ with an actual parameter of the type FILE OF <type> or TEXT, or the identical FCBFQQ type. The FCBFQQ type is the underlying record type that implements the file type in Wang PC Pascal. The interface for the target system FCBFQQ type (and any other types needed) is usually part of the internal file system. Thus, you can call procedures and functions that reference FCBFQQ parameters with any file type, including predeclared procedures and functions such as CLOSE and READ.

You can pass an FCBFQQ type variable to procedures like READLN and WRITELN that require a textfile. This permits, for example, calling the interface routines on the target operating system directly, working with mixtures of Wang PC Pascal and Wang PC FORTRAN (which share the file system interface but have special FCBFQQ fields), and other special file system activities.

Such activities require a sound knowledge of the file system. See Section 9.2 for a discussion of the file system interface and file control block.

CHAPTER 17

REFERENCE AND OTHER TYPES

The array, record, and set types discussed in Chapter 15 let you describe data structures whose form, size, and means of access are predetermined. The file type, described in Chapter 16, is a structure that varies in size but whose form and means of access are predetermined.

This chapter discusses reference types, which allow data structures that vary in size and form and whose means of access is particular to the programming problem involved. It also includes notes on PACKED types and procedural and functional types.

17.1 REFERENCE TYPES

A reference to a variable or constant is an indirect way to access it. The pointer type is an abstract type for creating, using, and destroying variables allocated from an area called the heap. The heap is a dynamically growing and shrinking region of memory the compiler allocates for pointer variables.

Wang PC Pascal provides two machine-oriented address types: one for addresses that require 16 bits, the other for addresses that require 32 bits. Pointers generally appear in trees, graphs, and list processing. Address types provide an interface to the hardware and operating system; their use is frequently unstructured, machine-specific, low-level, and unsafe. The following sections discuss both pointers and address types.

17.1.1 Pointer Types

A pointer type is a set of values that point to variables of a given type. The type of the variables pointed to is the "reference type." The compiler dynamically allocates all reference variables from the heap with the NEW procedure. The compiler normally allocates Pascal variables on the stack or at fixed locations.

The only actions you can perform on pointers are:

- You can assign them
- You can test them for equality and inequality with the two operators = and <>
- You can pass them as value or reference parameters
- You can dereference them with the up arrow (↑)

Every pointer type includes the pointer value NIL. Pointers frequently create list structures of records, as the following example shows:

```

TYPE
  TREETIP = ↑ TREE;
  TREE = RECORD
    VAL: INTEGER;
    {Value of TREE cell.}
    LEFT, RIGHT: TREETIP
    {Pointers to other TREETIP cells.}
    {Note recursive definition.}
  END;

```

Unlike most type declarations, the declaration for a pointer type can refer to a type of which it is itself a component. The declaration can also refer to a type the same TYPE section later declares, as in TREE and TREETIP in the previous example. Such a declaration is called a forward pointer declaration and permits recursive and mutually recursive structures. Because pointers appear so often in list structures, forward pointer declarations occur frequently.

The compiler checks for one ambiguous pointer declaration. If the previous example was in a procedure nested in another procedure that also declared a type TREE, the reference type of TREETIP could be either the outer definition or the one following in the same TYPE section. Wang PC Pascal assumes the TREE type intended is the one later in the same TYPE section and gives the warning "Pointer Type Assumed Forward". At the extend-level, a pointer can have a super array type as a referent type. The compiler then passes the actual upper bounds of the array to the NEW procedure to create a heap variable of the correct size. Forward pointer declarations of the super array type are illegal.

Wang PC Pascal conforms to the ISO requirement for strict compatibility between pointers. You cannot declare two pointers with different types and then assign or compare them, even if they happen to point to the same underlying type. The following example demonstrates this:

```

VAR PRA : ↑ REAL;
    PRE : ↑ REAL;
BEGIN PRA := PRE END; {This is illegal!}

```

Reference and Other Types

Programs usually contain only one type declaration for a pointer to a given type. In the TREETIP example, the type of LEFT and RIGHT could be TREE instead of TREETIP, but then you could not assign variables of type TREETIP to these fields. It is sometimes useful, however, to make sure that you do not use two classes of pointers together, even if they point to the same type.

For example, if you have a type RESOURCE kept in a list and you declare two types, OWNER and USER, of type RESOURCE, the compiler catches assignment of OWNER values to USER variables and vice versa and issues a warning message.

If the initialization checking switch is on, a newly created pointer has an uninitialized value. If the NIL checking switch is on, it tests pointer values for various invalid values. Invalid values include NIL, uninitialized values, references to already disposed heap items, or invalid heap reference values.

17.1.2 Address Types

As a system implementation language, Wang PC Pascal needs a method of creating, manipulating, and dereferencing actual machine addresses. The pointer type is only applicable to variables in the heap.

There are two kinds of addresses, relative and segmented. The keywords ADR and ADS refer to the relative address type and the segmented address type, respectively. As the following example shows, you use the keywords both as type clause prefixes and as prefix operators:

```
VAR INT_VAR : INTEGER;
    REAL_VAR : REAL;
    A_INT    : ADR OF INTEGER;
              {Declaration of ADR variable}
    AS_REAL  : ADS OF REAL;
              {Declaration of ADS variable}

BEGIN
  INT_VAR := 1;
              {Normal integer variable}
  REAL_VAR := 3.1415;
              {Normal real variable}
  A_INT    := ADR INT_VAR;
              {ADR used as operator}
  AS_REAL  := ADS REAL_VAR;
              {ADS used as operator}
  WRITELN (A_INT↑,AS_REAL↑)
              {Note use of up arrow to dereference}
              {the address types.}
              {Output is 1 and 3.1415.}
END.
```

Table 17-1 presents the characteristics of relative and segmented address types, as implemented for different machines.

Table 17-1. Relative and Segmented Machine Addresses

Machine	ADR	ADS
8080	16-bit absolute	Same as ADR
8086	16-bit default data segment offset	16-bit offset, 16-bit segment
Z8000 (unsegmented)	16-bit data absolute	Same as ADR
Z8000 (segmented)	Same as ADS	16-bit segment, 16-bit offset

See Appendix A for details specific to the Wang PC.

In Wang PC Pascal, you can declare a variable that is an address:

```
VAR X : ADR OF BYTE;
```

Then, with the following record notation, you can assign numeric values to the actual variable:

```
X.R := 16#FFFF
```

In an unsegmented environment, the .R (relative address) is the only record field available for ADR and ADS addresses.

Because Wang PC Pascal allows nondecimal numbering, you can specify the assigned value in hexadecimal notation. You can also assign to a segment field with the ADS type in a segmented environment, using the field notation .S (segment address). Thus, you can declare a variable of an ADS type and then assign values to its two fields:

```
VAR Y : ADS OF WORD;
```

```
  :
```

```
Y.S := 16#0001
```

```
Y.R := 16#FFFF
```

Reference and Other Types

As shown above, you can directly assign any 16-bit value to address type variables, using the .R and .S fields. The ADR and ADS operators obtain these addresses directly. The following example assigns addresses to the variables X and Y this way:

```

VAR X : ADR OF BYTE;
    Y : ADS OF WORD;
    W : WORD;
    B : BYTE;
    .
    X := ADR B;
    Y := ADS W;

```

Wang PC Pascal supports two predeclared address types:

```

ADRMEM = ADR OF ARRAY [0..32766] OF BYTE;
ADSMEM = ADS OF ARRAY [0..32766] OF BYTE;

```

Because the type the address refers to is an array of bytes, byte indexing is possible. For example, if A is of type ADRMEM, then A↑[15] is the byte at the address A.R + 15, where .R specifies an actual 16-bit address.

You can use the address types for a constant address (a form of structured constant); you can also take the address of a constant or expression. For example:

```

TYPE ADWORD = ADR OF WORD;
    ADSWORD = ADS OF WORD;
VAR W: WORD;
    R: ADWORD;
CONST CONADR = ADWORD (1234);
BEGIN
    W := CONADR↑;
    {Get word at address 1234}
    W := ADSWORD (0, 32)↑;
    {Get word at address 0:32}
    W := (ADS W).S;
    {Get value of DS segment register}
    R := ADR '123';
    {Get address of a constant value}
    R := ADR (W DIV 2 + 1);
    {Get address of expression value}
END;

```

However, it is illegal to use constants or expressions that yield addresses as the target of an assignment (or as a reference parameter or WITH record), as shown:

```
CONST ADSCON = ADSWORD (256, 64);    {OK}
FUNCTION SOME_ADDRESS: ADSWORD;      {OK}
BEGIN
  ADSWORD (0, 32)↑ := W; {Not permitted}
  ADSCON↑ := 12;         {Not permitted}
  SOME_ADDRESS↑ := 100; {Not permitted}
END;
```

17.1.3 Segment Parameters for the Address Types

Two keywords, VARS and CONSTS, are available as parameter prefixes (like VAR and CONST) to pass the segmented address of a variable. If P is of type ADS FOO, then you can pass P↑ to a VARS formal parameter, such as VARS X: FOO, but you cannot pass to a VAR formal parameter.

Segmented machine environments assume a default data segment. These environments pass a VAR parameter as the default data segment offset of a variable, and pass a VARS parameter as both the segment value and the offset value.

In the 8086 environment, both VARS parameters and ADS variables have the offset (.R) value in the WORD with the lower address and the segment (.S) value in the address plus two.

In the segmented Z8000 environment, the segment (.S) value is in the lower address and the offset (.R) value in the address plus two. The ADR type is also identical to the ADS type.

In the nonsegmented environment (such as 8080), VAR and CONST are the same as VARS and CONSTS. Because ADS and ADR are identical in a nonsegmented environment, the ADS type is useful in situations where the target environment may change. For example, in Wang PC Pascal, some primitive file system calls are declared with ADS parameters.

In pointer type declarations, the up arrow (↑) prefixes the type it points to; in program statements, it dereferences a pointer so that the compiler can assign or operate on the value it points to. The up arrow also dereferences ADR and ADS types in program statements.

The compiler performs component selection with the up arrow (↑) before the unary operators ADR or ADS. Because the up arrow (↑) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, a reference parameter, as well as in expressions. Since ADS and ADR are prefix operators, they appear only in expressions, where they apply only to a variable or constant or expression.

Reference and Other Types

Two pointer variables are compatible only if they have the same type; it is not enough that they point to the same type. Two address types are considered the same type, however, if they are both ADR or both ADS types. This lets you assign an ADR OF WORD to an ADR OF STRING (200). Such an assignment makes it easy to erase part of memory by assigning a variable of type STRING (200) to the 200 bytes that start at the address of a WORD variable.

If P1 is type ADR OF STRING (200) and P2 is any ADR OF type, the assignment $P1 \uparrow := P2 \uparrow$ generates fast code with no range-checking. Although this capability is not safe, operating systems and other software sometimes require it.

ADR and ADS are not compatible with each other, but the .R notation should overcome or reduce the problem.

17.1.4 Using the Address Types

You can combine and intermingle the two address types. The following example illustrates the rules that apply in a segmented environment:

```
VAR
  P: ADS OF DATA;
  {P is segmented address of type DATA.}
  Q: ADR OF DATA;
  {Q is relative address of type DATA.}
  X: DATA;
  {X is a variable of type DATA.}

BEGIN
  P := ADS X;
  {Assign the address of X to P.}
  X := P↑;
  {Assign to X the value pointed to by P.}
  P := ADS P↑;
  {Assign to P the address of the value whose}
  {address P points to. P is unchanged}
  {by this assignment.}
  Q := ADR X;
  {Assign the relative address of X to Q.}
  Q.R := (ADR X).R;
  {Assign the relative address of X to Q.}
  {using the WORD type.}
  P := ADS Q↑;
  {Assign address of variable at Q to P.}
  {You can always apply ADS to ADR↑.}
  Q := ADR P↑;
  {Illegal; you cannot apply ADR to ADS↑.}
  P.R := 16#8000;
  {Assign 32768 to P's offset field.}
  P.S := 16;
  {Assign 16 to P's segment field.}
```

```

Q.R := P.R + 4;
  {Assign P's offset plus 4 to be the value of Q.}
END;

```

See also the examples given in Section 17.1.2.

17.1.5 Notes on Reference Types

You should treat the address type and pointer type as two distinct types. The pointer type, in theory, is an undefined mapping from a variable to another variable. The method of implementation is undefined. However, the address type deals with actual machine addresses.

The following special facilities that use pointer variables are illegal with address variables:

- The NEW and DISPOSE procedures are only legal with pointers. NIL does not apply to the address type. No special address values exist for empty, uninitialized, or invalid addresses.
- Wang PC Pascal supports the type "address of super array type" in a way different from the "pointer to super array type." Getting the address of a super array variable is legal with ADR and ADS. For example, if a procedure or function formal parameter is VAR S: STRING, then the expression ADS S is legal within the procedure or function. Unlike a pointer, the address does not contain any upper bounds.

17.2 PACKED TYPES

Any of the structured types can be PACKED types. This can economize storage at the possible expense of access time or access code space. In Wang PC Pascal, however, some limitations on the use of PACKED structures apply:

- The compiler ignores the prefix PACKED, except for type checking, in sets, files, and arrays of characters; it has no actual effect on the representation of records and other arrays. Furthermore, PACKED can only precede one of the structure names ARRAY, RECORD, SET, or FILE; it cannot precede a type identifier. For example, if COLORMAP is the identifier for an unpacked array type, "PACKED COLORMAP" is not acceptable.
- You cannot pass a component of a PACKED structure as a reference parameter or use it as the record of a WITH statement unless the structure is of a string type. Obtaining the address of a PACKED component with ADR or ADS is illegal.

Reference and Other Types

- A PACKED prefix only applies to the structure you define; the prefix does not pack any components of that structure that are also structures unless you explicitly include the reserved word PACKED in their definition. Section 15.1 discusses the only exception to this rule, n-dimensional arrays.

17.3 PROCEDURAL AND FUNCTIONAL TYPES

Procedural and functional types are different from other Wang PC Pascal types. (For the remainder of this manual, the term "procedural" implies both procedural and functional). You cannot declare an identifier for a procedural type in a TYPE section, nor can you declare a variable of a type. You can use procedural types to declare the type of a procedural parameter, however, and in this sense they conform to the Pascal idea of a type.

A procedural type defines a procedure or function heading and gives any parameters. For a function, it also defines the result type. The syntax of a procedural type is the same as a procedure or function heading, including any attributes. Wang PC Pascal has no procedural variables, only procedural parameters. An example of a procedural type declaration is:

```
PROCEDURE ZERO (FUNCTION FUN (X, Y: REAL): REAL)
```

The compiler ignores parameter identifiers in a procedural type (X and Y in the previous example); only their type is important.

Section 22.4.3 provides more information about procedural types in Wang PC Pascal.

CHAPTER 18 CONSTANTS

18.1 INTRODUCTION

A constant is a value that is known before a program starts and that does not change as the program progresses. Examples of constants include the number of days in the week, your birth date, the name of your dog, or the phases of the moon.

You can give a constant an identifier, but you cannot alter the value associated with that identifier during the execution of the program. When you declare a constant, its identifier becomes a synonym for the constant itself.

Each constant implicitly belongs to some category of data:

- Numeric constants (discussed in Section 18.3) are one of the several number types: REAL, INTEGER, WORD, or INTEGER4.
- Character constants (discussed in Section 18.4) are strings of characters enclosed in single quotation marks and called "string literals" in Wang PC Pascal.
- Structured constants (discussed in Section 18.5) are also available at the extend-level. They include constant arrays, records, and typed sets.
- Constant expressions (discussed in Section 18.6) are also available at the extend-level. They let you compute a constant based on the values of previously declared constants in expressions.

The identifiers you define in an enumerated type are constants of that type. You cannot use them directly with numeric (or string) constant expressions. You can use these identifiers with the ORD, WRD, or CHR functions (For example, ORD (BLUE)). The extend-level also permits reading and writing the enumerated type's constant identifiers directly as character strings.

TRUE and FALSE are predeclared constants of type BOOLEAN; you can redeclare them. NIL is a constant of any pointer type; however, because it is a reserved word, you cannot redefine it. The null set is a constant of any set type.

Constants

Numeric statement labels have nothing to do with numeric constants; you cannot use a constant identifier or expression as a label. Internally, the maximum length for all constants is 255 bytes.

18.2 DECLARING CONSTANT IDENTIFIERS

Declaring a constant identifier introduces the identifier as a synonym for the constant. These declarations go in the CONST section of a compilant, procedure, or function.

The general format of a constant identifier declaration is the identifier followed by an equals sign and the constant value. The following program fragment includes three statements that identify constants (beginning after the word CONST):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
      DAYSINWEEK = 7;
      NAMEOFFPLANET = 'EARTH';
```

In this example, the numbers 365 and 7 are numeric constants. 'EARTH' is a string literal constant; you must enclose it in single quotation marks.

When you compile a program, the constant identifiers are not actually defined until after the compiler processes the declarations. Thus, a constant declaration such as the following has no meaning:

```
N = -N
```

The ISO standard defines a strict order in which to set out the declarations in the declaration section of a program:

```
CONST MAX = 10;
TYPE NAME = PACKED ARRAY [1..MAX] OF CHAR;
VAR FIRST : NAME;
```

Wang PC Pascal relaxes this order and, in fact, allows more than one instance of each kind of declaration:

```
TYPE COMPLEX = RECORD R, I : REAL END;
CONST PII = COMPLEX (3.1416, 00);
VAR PIX : COMPLEX;
TYPE IVEC = ARRAY [1..3] OF COMPLEX;
CONST PIVEC = IVEC (PII, PII, COMPLEX (0.0, 1.0));
```

18.3 NUMERIC CONSTANTS

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL, INTEGER, WORD, or INTEGER4.

Constants

Numbers can have a leading plus sign (+) or minus sign (-), except when the numbers are within expressions. Therefore:

```
ALPHA := +10      {Is legal}
ALPHA + -10      {Is illegal}
```

You cannot embed blanks within constants.

The compiler truncates any constant that exceeds 31 characters and gives a warning when this occurs.

The syntax for numeric constants applies not only to the actual text of programs, but also to the content of text files that a program reads.

The following are examples of numeric constants:

```
123      0.17      +12.345   007      -1.7E-10
-26.0    17E+3     26.0E12   -17E3     1E1
```

Numeric constants can appear in any of the following:

- CONST sections
- Expressions
- Type clauses
- Set constants
- Structured constants
- CASE statement CASE constants
- Variant record tag values

The following subsections discuss the different types of numeric constants in detail.

18.3.1 REAL Constants

The type of a number is REAL if the number includes a decimal point or exponent. The REAL value range depends on the REAL number unit of the target machine. Generally, this provides about seven digits of precision, with a maximum value of about 1.701411E38.

There is, however, a distinction between REAL values and REAL constants. The REAL constant range may be a subset of the REAL value range. REAL numeric constants must be greater than or equal to 1.0E-38 and less than 1.0E+38.

Constants

The compiler issues a warning if there is not at least one digit on each side of a decimal point. A REAL number that starts or ends with a decimal point can be misleading. For example, because left parenthesis-period substitutes for left square bracket and right parenthesis-period substitutes for right square bracket, the compiler interprets the expression `(.1+2.)` as `[1+2]`.

The compiler supports scientific notation in REAL numbers (as in `1.23E-6` or `4E7`). The decimal point and exponent sign are optional when you specify exponents. Both the uppercase E and the lowercase e are acceptable in REAL numbers. D and d can also indicate an exponent; this provides compatibility with other languages.

When you use IEEE REAL4 and REAL8 format, all real constants are stored in REAL8 (double-precision) format. If you require a single-precision REAL4 constant, declare a REAL4 variable and give it your REAL constant value in a VALUE section. (You may wish to give this variable the READONLY attribute as well.)

18.3.2 INTEGER, WORD, and INTEGER4 Constants

A numeric constant that is not type REAL is type INTEGER, WORD, or INTEGER4. Table 18-1 shows the range of values that constants of each of these types can assume.

Table 18-1. INTEGER, WORD, and INTEGER4 Constants

Type	Range of Values (minimum/maximum)	Predeclared Constant
INTEGER	-MAXINT to MAXINT	MAXINT=32767
WORD	0 to MAXWORD	MAXWORD=65535
INTEGER4	-MAXINT4 to MAXINT4	MAXINT4=2147483647

MAXINT, MAXWORD, and MAXINT4 are all predeclared constant identifiers.

One of three things happens when you declare a numeric constant identifier:

1. A constant identifier from -MAXINT to MAXINT becomes an INTEGER.
2. A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.
3. A constant identifier from -MAXINT4 to -MAXINT-1 or MAXWORD+1 to MAXINT4 becomes an INTEGER4.

Constants

Any INTEGER-type constant, including constant expressions and values from -32767 to -1, automatically changes to type WORD if necessary. If the INTEGER value is negative, the compiler adds 65536 to it and the underlying 16-bit value does not change. For example, you can declare a subrange of type WORD as WRD(0)..127; the upper bound of 127 becomes the type WORD. The reverse is not true; constants of type WORD do not automatically change to type INTEGER.

The ORD and WRD functions also change the type of an ordinal constant to INTEGER or WORD. Also, any INTEGER or WORD constant automatically changes to type INTEGER4 if necessary, but the reverse is not true. Table 18.2 presents examples of relevant conversions.

Table 18-2. Constant Conversions

Constant	Assumed Type
0	INTEGER could become WORD or INTEGER4
-32768	INTEGER4 only
32768	WORD could become INTEGER4
0..20000	INTEGER subrange
0..50000	WORD subrange
0..80000	Invalid: no INTEGER4 subranges
-1..50000	Invalid: becomes 65535..50000 (i.e., -1 is treated as 65536)

At the standard level, any numeric constant (that is, literal or identifier) can have a plus (+) or minus (-) sign.

18.3.3 Nondecimal Numbering

At the extend-level, Wang PC Pascal supports not only decimal number notation, but also numbers in hexadecimal, octal, binary, or other base numbering (where the base can range from 2 to 36). The number sign (#) acts as a radix separator. The following are examples of numbers in nondecimal notation:

16#FF02 10#987 8#776 2#111100

Constants

Wang PC Pascal recognizes leading zeros in the radix, so a number such as 008#147 is legal. In hexadecimal notation, uppercase or lowercase letters A through F are legal. A nondecimal constant without the radix (such as #44) is hexadecimal. Nondecimal notation does not imply a WORD constant; you can use it for INTEGER, WORD, or INTEGER4 constants. You cannot use nondecimal notation for REAL constants or numeric statement labels.

18.4 CHARACTER STRINGS

Most Pascal manuals refer to sequences of characters enclosed in single quotation marks as "strings." In Wang PC Pascal, they are called "string literals" to distinguish them from string constants, which can be expressions or values of the STRING type.

A string constant contains from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1..n] OF CHAR, also known in Wang PC Pascal as the type STRING (n). A string constant that contains just one character is of type CHAR. However, the type changes from CHAR to PACKED ARRAY [1..1] OF CHAR (for example, STRING (1)) if necessary. For example, you could assign a constant ('A') of type CHAR to a variable of type STRING (1).

Two adjacent single quotation marks represent literal apostrophe (single quotation mark) (for example, 'DON'T GO'). The null string (') is illegal. A string literal must fit on a line. The compiler recognizes string literals enclosed in double quotations marks ("), instead of single quotation marks, but issues a warning message when it encounters them.

The constant expression feature (discussed in Section 18.6) permits string constants made up of concatenations of other string constants, including string constant identifiers, the CHR () function, and structured constants of type STRING. This is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'THIS IS UNDERLINED' * CHR(13) * STRING (DO 18 OF '_')
```

The LSTRING feature of Wang PC Pascal adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [0..n] OF CHAR, except that element 0 contains the length of the string, which can vary from 0 to a maximum of 255 (see Section 15.2.2). If necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING.

NULL is a predeclared constant for the null LSTRING, with the element 0 (the only element) equal to CHR (0). You cannot concatenate NULL, since it is not of type STRING. It is the only constant of type LSTRING.

Constants

The following are examples of string literal declarations:

```
NAME = 'John Jacob';    {a legal string literal}
LETTER = 'Z';           {LETTER is of type CHAR}
QUOTED_QUOTE = ''';    {Quotes quote}
NULL_STRING = NULL;    {legal}
NULL_STRING = '';      {illegal}
DOUBLE = "OK";         {generates a warning}
```

18.5 STRUCTURED CONSTANTS

Standard Pascal permits only the numeric and string constants already mentioned, the pointer constant value NIL, and untyped constant sets. At the extend-level of Wang PC Pascal, however, you can use constant arrays, records, and typed sets. You can use structured constants anywhere a structured value is legal, in expressions as well as in CONST and VALUE sections.

- An array constant consists of a type identifier followed by a list of constant values in parentheses separated by commas. An example of an array constant is:

```
TYPE VECT_TYPE = ARRAY [-2..2] OF INTEGER;
CONST VECT = VECT_TYPE (5, 4, 3, 2, 1);
VAR A : VECT_TYPE;
VALUE A := VECT;
```

- A record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas. An example of a record constant is:

```
TYPE REC_TYPE = RECORD
  A, B: BYTE;
  C, D: CHAR;
END;
CONST RECR = REC_TYPE (#20, 0, 'A', CHR (20));
VAR FOO : REC_TYPE;
VALUE FOO := RECR;
```

- A set constant consists of an optional set type identifier followed by set constant elements in square brackets. Commas separate set constant elements. A set constant element is either an ordinal constant or two ordinal constants separated by two periods to indicate a range of constant values. An example of a set constant is:

```
TYPE COLOR_TYPE = SET OF
  (RED, BLUE, WHITE, GREY, GOLD);
CONST SETC = COLOR_TYPE (RED, WHITE .. GOLD);
VAR RAINBOW : COLOR_TYPE;
VALUE RAINBOW := SETC;
```

Constants

A constant within a structured array or record constant must have a type that the compiler can assign to the corresponding component type. For records with variants, the value of a constant element that corresponds to a tag field selects a variant, even if the tag field is empty. The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are legal.

An array or record constant nested within another structured constant must still have the preceding type identifier. For this reason, a super array constant can have only one dimension (see Section 15.2). The size of the representation of a structured constant must be from 1 to 255 bytes. If this 255-byte limit is a problem, declare a structured variable with the READONLY attribute, and initialize its components in a VALUE section.

An example of a complex structured constant follows:

```

TYPE R3 = ARRAY [1..3] OF REAL;
TYPE SAMPLE = RECORD I: INTEGER;
                    A: R3;
                    CASE BOOLEAN OF
                      TRUE: (S: SET OF 'A'..'Z';
                             P: ↑ SAMPLE);
                      FALSE: (X: INTEGER);
                    END;
CONST SAMP_CONST= SAMPLE (27, R3 (1.4, 1.4, 1.4),
                          TRUE, ['A','E','I'], NIL);

```

You can repeat constant elements with the phrase DO <n> OF <constant>, so the previous example could have used DO 3 OF 1.4 instead of 1.4, 1.4, 1.4.

Wang PC Pascal does not support set constant expressions such as [''] + LETTERS, or file constant expressions. The constant 'ABC' of type STRING (3) is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are illegal. Use the corresponding STRING constants instead.

The compiler can pass structured constants (and other structured values, such as variables and values returned from functions) by reference using CONST parameters. For more information, see Section 22.4.

Two kinds of set constants exist: one with an explicit type, as in CHARSET ['A'..'Z'], and one with an unknown type, as in [20..40]. You can use either in an expression or to define the value of a constant identifier. The compiler can also pass set constants with an explicit type as a reference (CONST) parameter. Sets of unknown type are unpacked, but the type changes to PACKED if necessary. Passing sets by reference is generally more efficient than passing them as value parameters.

Constants

18.6 CONSTANT EXPRESSIONS

Constant expressions in Wang PC Pascal allow you to compute constants based on the values of previously declared constants in expressions. Constant expressions can also occur within program statements. The following is an example of a constant expression declaration:

```
CONST HEIGHT_OF_LADDER = 6;
      HEIGHT_OF_MAN    = 6;
      REACH = HEIGHT_OF_LADDER + HEIGHT_OF_MAN;
```

Because a constant expression can contain only constants that you have declared earlier, the following is illegal:

```
CONST MAX = A + B;
      A   = 10;
      B   = 20;
```

You can use certain functions within constant expressions. For example:

```
CONST A = LOBYTE (-23) DIV 23;
      B = HIBYTE (-A);
```

Table 18-3 shows the functions and operators you can use with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

Table 18-3. Constant Operators and Functions

Type of Operand	Functions and Operators
REAL, INTEGER	Unary plus (+) Unary minus (-)
INTEGER, WORD	+ DIV OR HIBYTE() - MOD NOT LOBYTE() * AND XOR BYWORD()
Ordinal types	< <= CHR() LOWER() > >= ORD() UPPER() = <> WRD()
Boolean	AND NOT OR
ARRAY	LOWER() UPPER()
Any type	SIZEOF() RETYPE()

Constants

The following are examples of constant expressions:

```
CONST FOO = (100 + ORD('X')) * 8#100 + ORD('Y');
      MAXSIZE = 80;
      X = (MAXSIZE > 80) OR (IN_TYPE = PAPERTAPE);
      {X is a BOOLEAN constant}
```

In addition to the operators shown in Table 18-3 for numeric constants, you can use the string concatenation operator (*) with string constants, as follows:

```
CONST A = 'abcdef';
      M = CHR (109); {CHR is allowed}
      ATOM = A * 'ghijkl' * M;
      {ATOM = 'abcdefghijklm'}
```

These constants can span more than one line, though they cannot exceed the 255-character maximum. These string constant expressions are legal wherever a string literal is legal, except in metacommands.

Constants

CHAPTER 19

VARIABLES AND VALUES

19.1 INTRODUCTION

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type. A variable can have an identifier.

If A is a variable of type INTEGER, using A in a program actually refers to the data A denotes. For example:

```
VAR A: INTEGER;
BEGIN
  A := 1;
  A := A + 1;
END;
```

These statements first assign a value of 1 to the data A denotes, and subsequently assign it a value of 2.

You can manipulate variables by using notation to denote the variable; in the simplest case, this is a variable identifier. In other cases, you can denote variables by array indices or record fields or the dereferencing of pointer or address variables.

The compiler itself sometimes creates "hidden" variables, allocated on the stack, in circumstances such as the following:

- When you call a function that returns a structured result, the compiler allocates a variable in the caller for the result.
- When you need the address of an expression (for example, to pass it as a reference parameter or to use it as a WITH statement record or with ADR or ADS), the compiler allocates a variable for the value of the expression.
- The compiler may allocate a variable for the initial and final values of a FOR loop.

- When the compiler evaluates an expression, it may allocate a variable to store intermediate results.
- Every WITH statement requires the compiler to allocate a variable for the address of the WITH's record.

19.2 DECLARING A VARIABLE

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. Declare variables of the same type by giving a list of the variable identifiers, followed by their common type. For example:

```
VAR XCOORD, YCOORD: REAL
```

You can declare a variable in any of the following locations:

- VAR section of a program, procedure, function, module, interface, or implementation
- Formal parameter list of a procedure, function, or procedural parameter

In a VAR section, you can declare a variable to be of any legal type; in a formal parameter list, you can include only a type identifier. You cannot declare a type in the heading of a procedure or function). For example:

```
PROCEDURE NAME (GEORGE: ARRAY [1..10] OF COLOR)
{Illegal; GEORGE is of a new type.}

VAR VECTOR_A: VECTOR (10)
{Legal; VECTOR (10) is a type derived from}
{a super type.}
```

Each declaration of a file variable F of type FILE OF T implies the declaration of a buffer variable of type T, denoted by FT. At the extend level, a file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are F.TRAP, F.ERRS, F.MODE, and so on. See Sections 16.2 and 16.6 for further information on buffer variables and FCBFQQ fields, respectively.

19.3 THE VALUE SECTION

The VALUE section in Wang PC Pascal lets you give initial values to variables in a program, module, procedure, or function. You can also initialize the variable in an implementation, but not in an interface.

Variables and Values

The VALUE section can only include statically allocated variables; that is, it can only include variables declared at the program, module, or implementation level, or a variable with the STATIC or PUBLIC attribute. Variables with the EXTERN or ORIGIN attribute cannot occur in a VALUE section because the compiler does not allocate them.

The VALUE section can contain assignments of constants to entire variables or to components of variables. For example:

```
VAR ALPHA : REAL;
    ID   : STRING (7);
    I    : INTEGER;
```

```
VALUE
    ALPHA := 2.23;
    ID[1] := 'J';
    I     := 1;
```

Within a VALUE section, however, you cannot assign a variable to another variable. The last line in the following example is illegal, because I must be a constant:

```
CONTS MAX = 10;
VAR I, J : INTEGER;
VALUE I := MAX;
    J := I;
```

If the SROM metaccommand is off, loading the static data segment initializes variables. If the SROM metaccommand is on, the VALUE section generates an error message because ROM-based systems usually cannot statically initialize data.

19.4 USING VARIABLES AND VALUES

At the standard level of Wang PC Pascal, denotation of a variable may designate one of three things:

1. An entire variable
2. A component of a variable
3. A variable that a pointer references

A value can be any of the following:

- A variable
- A constant
- A function designator

- A component of a value
- A variable that a reference value references

At the extend-level, a function can also return an array, record, or set. You can use the same syntax you use for variables to denote components of the structures these functions return.

This feature also allows you to dereference a reference type that a function returns. You can only use the function designator as a value, however, not as a variable. For example, the following is illegal:

```
F (X, Y)↑ := 42;
```

At the extend-level, you can also declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type (see Section 18.6 for further discussion of this topic).

The following are examples of structured constant components:

```
TYPE REAL3 = ARRAY [1..3] OF REAL;
{an array type}
CONST PIES = REAL3 (3.14, 6.28, 9.42);
{an array constant}
.
.
X := PIES [1] * PIES [3];
{that is, 3.14 * 9.42}
Y := REAL3 (1.1, 2.2, 3.3) [2];
{that is, 2.2}
```

19.4.1 Components of Entire Variables and Values

At the standard level, a variable identifier denotes an entire variable. A variable, function designator, or constant denotes an entire value.

The identifier followed by a selector that specifies the component denotes a component of a variable or value. The form of a selector depends on the type of structure (array, record, file, or reference).

Indexed Variables and Values

The array variable or value, followed by an index expression, denotes a component of an array. The index expression must be assignment-compatible with the index type in the array type declaration. An index type must always be an ordinal type. The index itself must be enclosed in brackets following the array identifier. The following are examples of indexed variables and values:

```
ARRAY_OF_CHAR ['C']
{Denotes the Cth element.}
```

Variables and Values

'STRING CONSTANT' [6]
 {Denotes the 6th element, the letter 'G'.}

BETAMAX [12] [-3]
 BETAMAX [12,-3]
 {These two lines say the same thing.}

ARRAY_FUNCTION (A, B) [C, D]
 {Denotes a component of a two-dimensional array}
 {returned by ARRAY_FUNCTION (A, B). A and B are}
 {actual parameters.}

You can specify the current length of an LSTRING variable, LSTR, in either of two ways:

1. With the notation LSTR [0], to access the length as a CHAR component
2. With the notation LSTR.LEN, to access the length as a BYTE value

Field Variables and Values

The record variable or value followed by the field identifier for the component denotes a component of a record. Fields are separated by the period (.). In a WITH statement, you give the record variable or value only once. Within the WITH statement, you can use the field identifier of a record variable directly. The following are examples of field variables and values:

PERSON.NAME := 'PETE'

PEOPLE.DRIVERS.NAME := 'JOAN'

WITH PEOPLE.DRIVERS DO NAME := 'GERI'

RECURSING_FUNC ('XYZ').BETA
 {Selects BETA field of record returned}
 {by the function named RECURSIVE_FUNC.}

COMPLEX_TYPE (1.2, 3.14).REAL_PART

Record field notation also applies to files for FCBFQQ fields, to address type values for numeric representations, and to LSTRINGS for the current length.

File Buffers and Fields

Only one component of a file is accessible at any time. The current file position determines the accessible component and the buffer variable represents it. Depending on the status of the buffer variable, fetching its value may first read the value from the file. (This is called "lazy evaluation"; see Section 24.1.5 for details.)

If you pass a file buffer variable as a reference parameter or use it as a record of a WITH statement, the compiler issues a warning to alert you to the fact that the value of the buffer variable may not be correct after the position of the file changes with a GET or PUT procedure. The following are examples of file reference variables:

```
INPUT↑
ACCOUNTS_PAYABLE.FILE↑
```

19.4.2 Reference Variables

Reference variables or values denote data that refers to some data type. Three kinds of reference variables and values exist:

1. Pointer variables and values
2. ADR variables and values
3. ADS variables and values

In general, a reference variable or value "points" to a data object. Thus, the value of a reference variable or value is a reference to that data object. To obtain the actual data object pointed to, you must append an up arrow (↑) to the variable or value. The following is an example using pointer values:

```
VAR  P, Q : ↑INTEGER;
      {P and Q are pointers to integers.}

NEW (P); NEW (Q);
      {P and Q are assigned reference values to}
      {regions in memory that correspond to data}
      {objects of type INTEGER.}

P := Q;
      {P and Q now point to the same region}
      {in memory.}

P↑ := 123;
      {Assigns the value 123 to the INTEGER value}
      {P points to. Since Q points to this}
      {location as well, Q↑ is also assigned 123.}
```

Using NIL↑ is an error (because a NIL pointer does not reference anything). At the extend-level, you can also append an up arrow (↑) to a function designator for a function that returns a pointer or address type. In this case, the up arrow denotes the value the return value references. You cannot assign this variable to or pass it as a reference parameter.

Variables and Values

The following are examples of functions that return reference values:

```
DATA1 := FUNK1 (I, J)↑
{FUNK1 returns a reference value. The up arrow}
{dereferences the reference value returned,}
{assigning the referenced data to DATA1.}
```

```
DATA2 := FUNK2 (K, L)↑.FOO [2]
{FUNK2 returns a reference value. The up arrow}
{dereferences the reference value returned. In}
{this case, the dereferenced value is a record.}
{The array component FOO [2] of that record is}
{assigned to the variable DATA2.}
```

If P is of type ADR OF some type, then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address and P.S denotes the segment portion of the address. Both portions are of type WORD. The following are examples of address variables:

```
BUFF_ADR.R
DATA_AREA.S
```

19.5 ATTRIBUTES

At the extend-level of Wang PC Pascal, a variable declaration or the heading of a procedure or function may include one or more attributes. A variable attribute gives special information about the variable to the compiler. Table 19-1 displays the attributes Wang PC Pascal provides for variables.

Table 19-1. Attributes for Variables

Attribute	Variable
STATIC	Allocated at a fixed location, not on the stack
PUBLIC	Accessible by other modules with EXTERN, implies STATIC
EXTERN	Declared PUBLIC in another module, implies STATIC
ORIGIN	Located at specified address, implies STATIC
PORT	I/O address, implies STATIC
READONLY	Cannot be altered or written to

Variables and Values

The EXTERN attribute is also a procedure and function directive; PUBLIC and ORIGIN are also procedure and function attributes. See Section 22.3 for a discussion of procedure and function attributes and directives.

You can only give attributes for variables in a VAR section. Specifying variable attributes in a TYPE section or a procedure or function parameter list is illegal.

You give one or more attributes in the variable declaration, enclosed in brackets and separated by commas (if specifying more than one attribute). The brackets can occur in either of two places:

1. An attribute in brackets after a variable identifier in a VAR section applies to that variable only.
2. An attribute in brackets after the reserved word VAR applies to all of the variables in the section.

Examples that specify variable attributes are:

```
VAR A, B, C [EXTERN] : INTEGER;
{Applies to C only.}
```

```
VAR [PUBLIC] A, B, C : INTEGER;
{Applies to A, B, and C.}
```

```
VAR [PUBLIC] A, B, C [ORIGIN 16#1000] : INTEGER;
{A, B, and C are all PUBLIC. ORIGIN of C}
{is the absolute hexadecimal address 1000.}
```

19.5.1 The STATIC Attribute

The STATIC attribute gives a variable a unique fixed location in memory. This is in contrast to a procedure or function variable that the compiler allocates on the stack or dynamically allocates on the heap. STATIC variables can save time and code space, but they increase data space.

The compiler automatically assigns all variables at the program, module, or unit level a fixed memory location and gives them the STATIC attribute.

Functions and procedures that use STATIC variables can execute recursively, but you can use STATIC variables only for data common to all invocations. Because most of the other variable attributes imply the STATIC attribute, the tradeoff between savings in time and code space or reduced data space applies to the PUBLIC, EXTERN, ORIGIN, and PORT attributes as well.

Files the STATIC attribute declares in a procedure or function open when the routine is entered; they close when the routine terminates like other files. Other STATIC variables, however, are only open before program execution. This means that, except for open FILE variables, STATIC variables can retain values between invocations of a procedure or function.

Variables and Values

The following are examples of STATIC variable declarations:

```
VAR VECTOR [STATIC]: ARRAY [0..MAXVEC] OF INTEGER;
VAR [STATIC] I, J, K: 0..MAXVEC;
```

The STATIC attribute does not apply to procedures or functions, as some other attributes do.

19.5.2 The PUBLIC and EXTERN Attributes

The PUBLIC attribute indicates a variable that other loaded modules can access. The EXTERN attribute identifies a variable that resides in some other loaded module. These attributes generate a code object file that passes the identifier to the target Linker (where it the Linker can truncate it if it imposes a length restriction).

Variables with the PUBLIC or EXTERN attribute are implicitly STATIC. The following are examples of PUBLIC and EXTERN variable declarations:

```
VAR [EXTERN] GLOBE1, GLOBE2: INTEGER;
{The variables GLOBE1 and GLOBE2 are declared}
{EXTERN, meaning that they must be declared}
{PUBLIC in some other loaded module.}

VAR BASE_PAGE [PUBLIC, ORIGIN #12FE]: BYTE;
{The variable BASE_PAGE is located at 12FE,}
{hexadecimal. Because it is also PUBLIC, it can}
{be accessed from other loaded modules that}
{declare BASE_PAGE with the EXTERN attribute.}
```

The compiler usually allocates PUBLIC variables, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the loader that a global name has an absolute address. You cannot combine PUBLIC with PORT.

If both PUBLIC and ORIGIN are present, the compiler does not need the loader to resolve the address. However, the compiler still passes the identifier to the Linker for use by other modules.

The compiler does not allocate EXTERN variables. These variables do not have an ORIGIN, since giving both EXTERN and ORIGIN implies two different ways to access the variable. The reserved word EXTERNAL is synonymous with EXTERN. This increases compatibility with other Pascals; all versions commonly use one of the two.

Variables in the interface of a unit automatically receive either the PUBLIC or EXTERN attribute. If a program, module, or unit USES an interface, its variables are made EXTERN; if you compile the IMPLEMENTATION of the interface, its variables are made PUBLIC.

19.5.3 The ORIGIN and PORT Attributes

The ORIGIN attribute directs the compiler to locate a variable at a given memory address; the PORT attribute specifies some kind of I/O address. In either case, the address must be a constant of any ordinal type. ORIGIN or PORT variables can access I/O ports, interrupt vectors, operating system data, and other related data. The following are examples of ORIGIN and PORT variable declarations:

```
VAR KEYBOARDP [PORT 16#FFF2]: CHAR;
VAR INTRVECT [ORIGIN 8#200]: WORD;
```

Variables with ORIGIN or PORT attributes are implicitly STATIC. They also inhibit common subexpression optimization. For example, if GATE has the ORIGIN attribute, the two statements `X := GATE; Y := GATE` access GATE twice in the order given, instead of using the first value for both assignments. This ensures correct operation if GATE is a memory-mapped input port. You cannot pass PORT variables as reference parameters.

The compiler never allocates or initializes ORIGIN and PORT variables. The associated address only indicates where the variable is located in memory. ORIGIN always implies a memory address, but the meaning of PORT varies with the implementation. For more information on the PORT attribute, see Appendix A.

Giving the PORT and ORIGIN attributes in brackets immediately following the VAR keyword is ambiguous and generates an error during compilation. (This is because it would be unclear to the compiler whether all variables following should be at the same address or whether it should assign addresses sequentially.)

```
VAR [ORIGIN 0] FIRST, SECOND: BYTE; {ILLEGAL!}
```

ORIGIN (but not PORT) permits a segmented address using segment: offset notation.

```
VAR SEGVECT [ORIGIN 16#0001:16#FFFE]: WORD;
```

You cannot use a variable with a segmented ORIGIN as the control variable in a FOR statement.

19.5.4 The READONLY Attribute

The READONLY attribute prevents assignments to a variable. It also prevents passing of the variable as a VAR or VARS parameter. In addition, a READ statement cannot read a READONLY variable, and you cannot use it as a FOR control variable. You can use READONLY with any of the other attributes. The following are examples of READONLY variable declarations:

Variables and Values

```
VAR IMPORT [PORT 12, READONLY]: BYTE;
{IMPORT is a READONLY PORT variable.}
```

```
VAR [READONLY] I, J [PUBLIC], K [EXTERN]: INTEGER;
{I, J, and K are all READONLY;}
{J is also PUBLIC; K is also EXTERN.}
```

CONST and CONSTS parameters, as well as FOR loop control variables (while in the body of the loop), receive the READONLY attribute automatically. READONLY is the only variable attribute that does not imply STATIC allocation.

A variable that is both READONLY and either PUBLIC or EXTERN in one source file is not necessarily READONLY when another source file uses it. The READONLY attribute does not apply to procedures and functions.

19.5.5 Combining Attributes

You can give a variable multiple attributes. Separate the attributes with commas and enclose the list in brackets, as shown:

```
VAR [STATIC]
X, Y, Z [ORIGIN #FFFE, READONLY]: INTEGER;
```

In this example, Z is a STATIC, READONLY variable with an ORIGIN at hexadecimal FFFE. These rules apply when you are combining attributes:

- If you give a variable the EXTERN attribute, you cannot give it the PORT, ORIGIN, or PUBLIC attributes in the current compilant.
- If you give a variable the PORT attribute, you cannot give it the ORIGIN, PUBLIC, or EXTERN attributes at all.
- If you give a variable the ORIGIN attribute, you cannot also give it the PORT or EXTERN attributes. However, you can combine ORIGIN with PUBLIC.
- If you give a variable the PUBLIC attribute, you cannot also give it the PORT or EXTERN attributes. However, you can combine PUBLIC with ORIGIN.
- You can use STATIC and READONLY with any other attributes.

CHAPTER 20

EXPRESSIONS

Expressions are constructions that the compiler evaluates. Table 20-1 illustrates a variety of expressions, which, if $A = 1$ and $B = 2$, evaluate to the value shown.

Table 20-1. Expressions

Expression	Value
2	2
A	1
A + 2	3
(A + 2)	3
(A + 2) * (B - 3)	-3

The operands in an expression can be a value or any other expression. When you apply any operator to an expression, that expression is called an operand. With parentheses for grouping and operators that use other expressions, you can construct expressions that are as long and complicated as you wish.

The available operators, in the order in which the compiler applies them, are as follows (operators shown in parentheses are available only at the extend-level of Wang PC Pascal, those in brackets only at the system level):

1. Unary: NOT, [ADR, ADS]
2. Multiplying: *, /, DIV, MOD, AND, (ISR, SHL, SHR)
3. Adding: +, -, OR, (XOR)
4. Relational: =, <>, <=, >=, < >, IN

In cases of ambiguity, the compiler applies an operator at a higher level before one at a lower level. For example, the expression $1 + 2 * 3$ evaluates to 7 instead of 9. Use parentheses to change operator precedence; thus, the expression $(1 + 2) * 3$ evaluates to 9 rather than 7.

If the `SSIMPLE` switch is on, the compiler executes sequences of operators of the same precedence from left to right. If the switch is off, the compiler may rearrange expressions and evaluate common subexpressions only once, in order to generate optimized code. The semantics of the precedence relationships remain, but normal associative and distributive laws apply. For example, $X * 3 + 12$ is an optimization of $3 * (6 + (X - 2))$.

These optimizations occasionally give you unexpected overflow errors. For example, the compiler optimizes the expression $(I - 100) + (J - 100)$ into the expression $(I + J) - 200$. This may result in an overflow error, although the original expression did not (for example, if I and J are each 16400).

The compiler may or may not actually evaluate an expression in your source file when the program runs. For example, the expression $F(X + Y) * 0$ is always zero, so the compiler does not execute the subexpression $(X + Y)$ and the function call. The compiler always evaluates expressions; the only exceptions are those stated in the following pages.

A Pascal expression is either a value or the result of applying an operator to one or two values. Although a value can be of almost any type, most Wang PC Pascal operators only apply to the following types:

INTEGER	INTEGER4	WORD
BOOLEAN	REAL	SET

The relational operators also apply for the `CHAR`, enumerated, string, and reference types. For all operators (except the set operator `IN`), operands must have compatible types.

21.1 SIMPLE TYPE EXPRESSIONS

As a rule, the operands and the value resulting from an operation are all of the same type. Occasionally, however, the type of an operand changes to the type an operator requires. This conversion occurs on two levels: one for constant operands only, and one for all operands. `INTEGER`-to-`WORD` conversion occurs for constant operands only; conversion from `INTEGER` to `REAL` and from `INTEGER` or `WORD` to `INTEGER4` occurs for all operands.

If necessary in constant expressions, `INTEGER` values change to `WORD` type. Be careful when mixing `INTEGER` and `WORD` constants in expressions. For example, if `CBASE` is the constant `16#C000` and `DELTA` is the constant `-1`, the expression `WRD (CBASE) + DELTA` gives a `WORD` overflow. The overflow occurs because the compiler converts `DELTA` to the `WORD` value `16#FFFF`, and `16#C000` plus `16#FFFF` is greater than `MAXWORD`. However, the expression `WRD (ORD (CBASE) + DELTA)` works. This expression gives the `INTEGER` value `-16385`, which changes to `WORD 16#BFFF`. If an operator needs a conversion or it is necessary for an assignment, the compiler makes the following conversions:

Expressions

- From INTEGER to REAL or INTEGER4
- From WORD to INTEGER4

The following rules determine the type of the result of an expression involving these simple types:

1. + - *

These operators operate on INTEGERS, REALs, WORDs, and INTEGER4s, as shown in the following examples:

```
+123
A + 123
-23.4
A - 8
A * B * 3
```

Mixtures of REALs with INTEGERS and of INTEGER4s with INTEGERS or WORDs are legal. Where both operands are of the same type, the result type is the type of the operands. If either operand is REAL, the result type is REAL; otherwise, if either operand is INTEGER4, the result type is INTEGER4.

Unary plus (+) and minus (-) are legal, as are the binary forms. Unary minus on a WORD type is two's complement (NOT is one's complement); because negative WORD values do not exist, this always generates a warning.

Because unary minus has the same precedence level as the adding operators,

```
(X * -1)      {Is illegal}
(-256 AND X)  {Is interpreted as -(256 AND X)}
```

2. /

This is a "true" division operator. The result is always REAL. Operands can be INTEGER or REAL (not WORD or INTEGER4). The following are examples of division:

```
34 / 26.4 = 1.28787...
18 / 6    = 3.00000...
```

3. DIV MOD

These are the operators for integer divide quotient and remainder, respectively. The left operand (dividend) is divided by the right operand (divisor). The following are examples of integer division:

```

123 MOD 5 = 3
-123 MOD 5 = -3 {Sign of result is}
                  {sign of dividend }
123 MOD -5 = 3
1.3 MOD 5      {Illegal with REAL operands}
123 DIV 5 = 24
1.3 DIV 5      {Illegal with REAL operands}

```

Both operands must be of the same type: INTEGER, WORD, or INTEGER4 (not REAL). The sign of the remainder (MOD) is always the sign of the dividend.

Wang PC Pascal differs from the current draft ISO standard with respect to the semantics for DIV and MOD with negative operands, but the resulting code is more efficient.

4. AND OR XOR NOT

These extend-level operators are bitwise logical functions. Operands must be INTEGER or WORD or INTEGER4 (never a mixture), and cannot be REAL. The result has the type of the operands.

NOT is a bitwise one's complement operation on the single operand. If an INTEGER variable V has the value MAXINT, NOT V gives the illegal INTEGER value -32768. This generates an error if the initialization switch is on and a program uses the value later.

Given the following initial INTEGER values,

```

X = 2#1111000011110000
Y = 2#1111111100000000

```

AND, OR, XOR, and NOT perform the following functions:

```

X AND Y    1111000011110000
            1111111100000000
            -----
            1111000000000000

X OR Y     1111000011110000
            1111111100000000
            -----
            1111111111110000

X XOR Y     1111000011110000
            1111111100000000
            -----
            0000111111110000

NOT X       1111000011110000
            -----
            0000111100001111

```

5. SHL SHR ISR

These extend-level operators provide bitwise shifting functions.

SHL and SHR are logical shifts left and right. ISR is an integer (signed) arithmetic shift right; it always propagates the sign bit, even on a WORD type operand. Because the compiler cannot generate a simple right shift for INTEGER division (-1 DIV 2 is incorrect) and division is a very time-consuming operation, you can use SHR or ISR instead of DIV where appropriate.

Operands must be both INTEGER, both WORD, or both INTEGER4; they cannot be REAL. The result has the same type as the operands.

The left operand is a shifted operand, and the right operand is the shift count in bits. A shift count less than 0 or greater than 31 produces undefined results and generates an error message if the range-checking switch is on. Shifts never cause overflow errors; they truncate shifted bits. If $X = 2\#1111111100000000$, the shifting functions perform the following operations:

```

X          1111111100000000
X SHL 1    1111111100000000
X SHR 1    0111111110000000
X ISR 1    1111111110000000 {sign extension}

```

20.2 BOOLEAN EXPRESSIONS

The Boolean operators at the standard level of Wang PC Pascal are:

NOT	AND	OR
=	<	>
<>	<=	>=

XOR is available at extend-level and system-level.

You can also use $P \langle \rangle Q$ as an exclusive OR function. Because $\text{FALSE} \langle \text{TRUE}, P \leq Q$ denotes the Boolean operation "P implies Q." Furthermore, the Boolean operators AND and OR are not the same as the WORD and INTEGER operators of the same name that are bitwise logical functions. The Boolean AND and OR operators may or may not evaluate their operations. The following example illustrates the danger of assuming that they do not:

```

WHILE (I <= MAX) AND (V [I] <> T) DO I := I + 1;

```

If array V has an upper bound MAX, the evaluation of V [I] for I > MAX is a runtime error. This evaluation may or may not take place. Sometimes the compiler evaluates both operands during optimization, and sometimes the evaluation of one causes the compiler to skip the other. In the latter case, the compiler may evaluate either operand first. Instead, use the following construction:

```
WHILE I <= MAX DO
  IF V [I] <> T THEN I := I + 1 ELSE BREAK;
```

See Section 21.3.5 for information on using AND THEN and OR ELSE to handle situations, such as the previous example, that examine tests sequentially.

The relational operators produce a Boolean result. The types of the operands of a relational operator (except for IN) must be compatible. If they are not compatible, one must be REAL and the other must be compatible with INTEGER.

You can only compare reference types with = and <>. To compare an address type with one of the other relational operators, you must use address field notation, as shown:

```
IF (A.R < B.R) THEN <statement>;
```

Except for the string types STRING and LSTRING, you cannot compare files, arrays, and records as wholes. You can compare two STRING types only if they have the same upper bound; two LSTRINGS can have different upper bounds.

An LSTRING comparison ignores characters past the current length. If the current length of one LSTRING is less than the length of the other and all characters up to the length of the shorter are equal, the compiler assumes the shorter one is "less than" the longer one. However, two LSTRINGS are not considered equal unless all current characters are equal and their current lengths are equal.

The six relational operators (=, <>, <=, >=, <, and >) have their normal meaning with numeric, enumerated, CHAR, or string operands. Section 20.3 discusses the meaning of these relational operators (along with the relational operator IN) with sets. Since the relational operators in Boolean expressions have a lower precedence than AND and OR, the following is incorrect:

```
IF I < 10 AND J = K THEN
```

Instead, you must write:

```
IF (I < 10) AND (J = K) THEN
```

Also, you cannot use the numeric types where a Boolean operand is required. (Some other languages permit this.) For an integer I, the clause IF I THEN is illegal; you must use the following instead:

```
IF I <> 0 THEN
```

Expressions

Wang PC Pascal does, however, allow the following:

\$IF I \$THEN

The inclusion of special not-a-number (NaN) values means that a comparison between two real numbers can have a result other than less-than, equal, or greater-than. The numbers can be unordered, meaning one or both are NaNs. An unordered result is the same as "not equal, not less than, and not greater than." For example, if variables A and/or B are NaN values:

- A < B is false.
- A <= B is false.
- A > B is false.
- A >= B is false.
- A = B is false.
- A <> B is, however, true.

REAL comparisons do not follow the same rules as other comparisons in many ways. A < B is not always the same as NOT (B <= A); this prevents some optimizations the compiler otherwise does. If A is a NaN, then A <> A is true; in fact, this is a good way to check for a NaN value.

20.3 SET EXPRESSIONS

Table 20-2 shows the Wang PC Pascal operators that apply differently to sets than to other types of expressions.

Table 20-2. Set Operators

Operator	Meaning in Set Operations
+	Set union
-	Set difference
*	Set intersection
=	Test set equality
<>	Test set inequality
<= and >=	Test subset and superset
< and >	Test proper subset and superset
IN	Test set membership

Any operand whose type is SET OF S, where S is a subrange of T, is operated on as if it were SET OF T. (T must be in the range from 0 to 255 or the equivalent ORD values.) Either both operands must be PACKED or neither must be PACKED, unless one operand is a constant or constructed set.

With the IN operator, the left operand (an ordinal) must be compatible with the base type of the right operand (a set). The expression X IN B is TRUE if X is a member of the set B, and FALSE otherwise. X can be outside of the range of the base type of B legally. For example, X IN B is always false if the following statements are true:

```
X = 1
B = SET OF 2..9
```

(1 is compatible, but not assignment-compatible, with 2..9.)

Angle brackets are set operators only at the extend-level of Wang PC Pascal, since the ISO standard does not support them for sets. They test that a set is a proper subset or superset of another set. Proper subsetting does not permit a set as a subset if the two sets are equal.

Expressions that involve sets may use the "set constructor," which gives the elements in a set enclosed in square brackets. Each element can be an expression whose type is in the base type of the set or the lower and upper bounds of a range of elements in the base type. Elements cannot be sets themselves. The following are examples of sets that involve set constructors:

```
SET_COLOR := [RED, BLUE..PURPLE] - [YELLOW]

SET_NUMBER :=
  [12, J+K, TRUNC (EXP (X))..TRUNC (EXP (X+1))]
```

Set constructor syntax is similar to CASE constant syntax. If $X > Y$, then $[X..Y]$ denotes the empty set. Empty brackets also denote the empty set and are compatible with all sets. In addition, if all elements are constant, a set constructor is the same as a set constant.

As is true of other structured constants, the type identifier for a constant set can appear in a set constant, as in COLORSET [RED..BLUE]. This does not mean that a set constructor with variable elements can receive a type in an expression: NUMBERSSET [I..J] is illegal if I or J is a variable.

A set constructor such as [I, J..K] or an untyped set such as [1, 5..7], is compatible with either a PACKED or an unpacked set. A typed set constant, such as DIGITS [1, 5..7], is only compatible with sets that are PACKED or unpacked, respectively, in the same way as the explicit type of the constant.

Expressions

20.4 FUNCTION DESIGNATORS

A function designator activates a function. It consists of the function identifier, followed by a (possibly empty) list of "actual parameters" in parentheses:

```
{Declaration of the function ADD.}
FUNCTION ADD (A, B: INTEGER); INTEGER;
.
.
{Use of the function ADD in an expression.}
X := ADD (7, X * 4) + 123;
{ADD is function designator.}
```

These actual parameters substitute, position for position, for their corresponding "formal parameters," defined in the function declaration.

Parameters can be variables, expressions, procedures, or functions. If the parameter list is empty, you must omit the parentheses. (See Section 22.4 for more information on parameters.)

The order of evaluation and binding of the actual parameters varies, depending on the optimizations that occur. If the \$SIMPLE metaccommand is on, the order is left to right.

In most computer languages, functions have two different uses:

1. In the mathematical sense, they take one or more values from a domain to produce a resulting value in a range. In this case, if the function never does anything else (such as assign to a global variable or perform input/output), it is called a "pure" function.
2. The second type of function can perform other tasks, such as changing a static variable or a file. Functions of this second kind are "impure."

At the standard level, a function can return either a simple type or a pointer. At the extend-level, a function can return any assignable type (any type except a file or super array).

At the standard level, the compiler can only compare, assign, or pass a pointer that a function returns as a value parameter. At the extend-level, however, the usual selection syntax for reference types, arrays, and records is legal, following the function designator. See Section 19.4 for information. The following are examples of function designators:

SIN (X+Y)

NEXTCHAR

Expressions

```

NEXTREC (17) ↑
{Here the function return type
{is a pointer, and the returned
{pointer value is dereferenced.}

NAD.NAME [1]
{Here the function has no parameters.}
{The return type is a record, one
{field of which is an array.}
{The identifier for that field is}
{NAME. The example above selects}
{the first array component of the}
{returned record.}

```

It is more efficient to return a component of a structure than to return a structure and then use only one component of it. The compiler treats a function that returns a structure like a procedure, with an extra VAR parameter representing the result of the function. The function's caller allocates an unseen variable (on the stack) to receive the return value, but the compiler only allocates this "variable" during execution of the statement that contains the function invocation.

20.5 EVALUATING EXPRESSIONS

The compiler can pass any expression as a CONST or CONSTS parameter or find the expression's address. It calculates and stores the expression in a temporary variable on the stack, and the address of this temporary variable can be a reference parameter or appear in some other address context.

To avoid ambiguities, enclose such an expression with operators or function calls in parentheses. For example, to invoke a procedure, you must use FOO (CONST X, Y: INTEGER), FOO (I, (J+14)) instead of FOO (I, J+14).

This implies a subtle distinction in the case of functions. For example:

```

FUNCTION SUM (CONST A, B: INTEGER): INTEGER;
BEGIN
  SUM := A;
  IF B <> 0 THEN
    SUM := SUM (SUM, (SUM (B, 0) - 1)) + 1;
  END;

```

This example calls SUM recursively, subtracting one from B until B is zero.

The use of a function identifier in a WITH statement follows a similar rule. For example, given a function without parameters (COMPLEX) that returns a record, "WITH COMPLEX" means "WITH the current value of the function." This can only occur inside the COMPLEX function itself. However, "WITH (COMPLEX)" calls the function and assigns the result to a temporary local variable.

Expressions

Another way to describe this is to distinguish between "address" and "value" phrases. The left side of an assignment, a reference parameter, the ADR and ADS operators, and the WITH statement all need an address. The right side of an assignment and a value parameter all need a value.

If the compiler needs an address but only a value, such as a constant or an expression in parentheses, is available, the compiler must put the value into memory so it has an address. For constants, the value goes in static memory; for expressions, the value goes in stack (local) memory. A function identifier refers to the current value of the function as an address, but calls the function as a value.

Finally, in the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same: the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses, as in (F(X)), forces evaluation of the function.

20.6 OTHER FEATURES OF EXPRESSIONS

EVAL and RESULT are two procedures available at the extend-level for use with expressions. EVAL obtains the effect of a procedure from a function; RESULT yields the current value of a function within a function or nested procedure or function. At the system level, the RETYPE function allows you to change the type of a value.

20.6.1 The EVAL Procedure

EVAL evaluates its parameters without actually calling anything. Generally, you use EVAL to obtain the effect of a procedure from a function. In such cases, the values functions return are of no interest, so EVAL is only useful for functions with side effects. For example, EVAL might call a function that advances to the next item and also returns the item just to advance to the next item, since there is no need to obtain a function return value. The following are examples of the EVAL procedure:

```
EVAL (NEXTLABEL (TRUE))
EVAL (SIDEFUNC (X, Y), INDEX (4), COUNT)
```

20.6.2 The RESULT Function

Within the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same: that is, the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses, as in (F (X)), forces evaluation of the function. The following are examples of the RESULT function:

```

FUNCTION FACTORIAL (I: INTEGER): INTEGER;
BEGIN
  FACTORIAL := 1; WHILE I > 1 DO
  BEGIN
    FACTORIAL := I * RESULT (FACTORIAL);
    I := I - 1;
  END;
END;

FUNCTION ABSVAL (I: INTEGER): INTEGER;
BEGIN
  ABSVAL := I;
  IF I < 0 THEN ABSVAL := -RESULT (ABSVAL);
END;

```

20.6.3 The RETYPE Function

Occasionally, you need to change the type of a value. You can do this with the RETYPE function, available at the system level of Wang PC Pascal. If the new type is a structure, the usual selection syntax can follow RETYPE. You must use RETYPE with caution: it works on the memory byte level and ignores whether the low order byte of a two-byte number comes first or second in memory.

The following are examples of the RETYPE function:

```

RETYPE (COLOR, 3)           {inverse of ORD}
RETYPE (STRING2, I*J+K) [2] {effect may vary}

```

CHAPTER 21 STATEMENTS

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute. This chapter first discusses the syntax of statements and then separates and describes two categories of statements: simple statements and structured statements. A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements. Table 21-1 lists the statements in each category in Wang PC Pascal.

Table 21-1. Wang PC Pascal Statements

Simple	Structured
Assignment (:=)	Compound
Procedure	IF/THEN/ELSE
GOTO	CASE
BREAK	FOR
CYCLE	WHILE
RETURN	REPEAT
Empty	WITH

21.1 THE SYNTAX OF PASCAL STATEMENTS

Pascal statements are separated by a semicolon (;) and enclosed by reserved words such as BEGIN and END. A statement begins, optionally, with a label. The following subsections discuss each of these three elements of statement syntax.

21.1.1 Labels

Any statement a GOTO statement refers to must have a label. A label at the standard level is one or more digits; the compiler ignores leading zeros. Constant identifiers, expressions, and nondecimal notation cannot serve as labels. You must declare all labels in a LABEL section. At the extend-level, a label can also be an identifier.

The following is an example using labels and GOTO statements:

```
PROGRAM LOOPS(INPUT,OUTPUT);
LABEL 1, HAWAII, MAINLAND;

BEGIN
  MAINLAND: GOTO 1;
  HAWAII: WRITELN ('Here I am in Hawaii');
  1: GOTO HAWAII
END.
```

A loop label is any label that immediately precedes a looping statement: WHILE, REPEAT, or FOR. At the extend-level, a BREAK or CYCLE statement can also refer to a loop label.

Both a CASE constant list and a GOTO label can precede a statement, in which case the CASE constants come first and then the GOTO label. In the following example, 321 is a CASE value, 123 is label:

```
321: 123: IF LOOP THEN GOTO 123
```

21.1.2 Separating Statements

Semicolons separate statements. Semicolons do not terminate statements. Because Pascal permits the empty statement, however, using the semicolon as if it were a statement terminator is permissible.

The following is an example of using semicolons to separate statements:

```
BEGIN
  10: WRITELN;
  A := 2 + 3;
  GOTO 10
END
```

A common error is to terminate the THEN clause in an IF/THEN/ELSE statement with a semicolon. For example, the following example generates a warning message:

```
IF A = 2 THEN WRITELN;
ELSE A = 3
```

Another common error is to put a semicolon after the DO in a WHILE or FOR statement:

```
FOR I := 1 TO 10 DO;
BEGIN
  A[I] := I;
  B[I] := 10 - I;
END;
```

Statements

The previous example executes an empty statement 10 times, then executes the array assignments once. As occasional legitimate uses for repeating an empty statement exist, no warning occurs when this happens.

The semicolon also follows the reserved word END at the close of a block of program statements.

21.1.3 The Reserved Words BEGIN and END

Whenever you want a program to execute a group of statements instead of a single simple statement, you can enclose the block with the reserved words BEGIN and END.

For example, the following group of statements between BEGIN and END execute if the condition in the IF statement is TRUE:

```
IF (MAX > 10) THEN
BEGIN
  MAX = 10;
  MIN = 0;
  Writeln (MAX,MIN)
END;
Writeln ('done')
```

At the extend-level, you can substitute a pair of square brackets for the keywords BEGIN and END.

21.2 SIMPLE STATEMENTS

A simple statement is one in which no part constitutes another statement. Simple statements in standard Pascal are:

- The assignment statement
- The procedure statement
- The GOTO statement
- The empty statement

The empty statement contains no symbols and denotes no action. It is included in the definition of the language primarily to permit you to use a semicolon after the last in a group of statements enclosed between BEGIN and END.

The extend-level in Wang PC Pascal adds three simple statements: BREAK, CYCLE, and RETURN.

11.2.1 Assignment Statements

The assignment statement replaces the current value of a variable with a new value, which you specify as an expression. An adjacent colon and equals sign characters (:=) denote assignment.

The following are examples of assignment statements:

```
A := B

A[I] := 12 * 4 + (B * C)

X := Y
{Illegal. Colon (:) and equals}
{sign (=) must be adjacent.}

A + 2 := B
{Illegal. A + 2 is not a variable.}

A := ADD (1,1)
```

The value of the expression must be assignment-compatible with the type of the variable. Selection of the variable may involve indexing an array or dereferencing a pointer or address. If it does, the compiler may, depending on the optimizations performed, mix these actions with the evaluation of the expression. If the SSIMPLE metacommand is on, the compiler evaluates the expression first.

An assignment to a nonlocal variable (including a function return) puts an equals sign (=) or percent sign (%) in the G column of the listing file. (See Section 26.5 for more information about these and other symbols used in the listing.)

Within the block of a function, an assignment to the identifier of the function sets the value the function returns. The assignment to a function identifier can occur either within the actual body of the function or in the body of a procedure or function nested within it.

If the range-checking switch is on, an assignment to a set, subrange, or LSTRING variable may imply a runtime call to the error-checking code.

According to the Wang PC Pascal optimizer, each section of code without a label or other point that can receive control is eligible for rearrangement and common subexpression elimination. Naturally, the order of execution remains the same when necessary.

Given these statements:

```
X := A + C + B;
Y := A + B;
Z := A
```

the compiler might generate code to perform the following operations:

Statements

- Get the value of A and save it.
- Add the value of B and save the result.
- Add the value of C and assign it to X.
- Assign the saved A + B value to Y.
- Assign the saved A value to Z.

This optimization occurs only if assignment to X and Y and getting the value of A, B, or C are all independent. If C is a function without the PURE attribute and A is a global variable, evaluating C may change A. Then, since the order of evaluation within an expression in this case is unknown, the value of A in the first assignment could be the old value or the new one. Because the order of evaluation among statements is known, however, the value of A in the second and third assignments is the new value.

The following actions may limit the ability of the optimizer to find common subexpressions:

- Assignment to a nonlocal variable
- Assignment to a reference parameter
- Assignment to the referent of a pointer
- Assignment to the referent of an address variable
- Calling a procedure
- Calling a function without the PURE attribute

The optimizer does allow for "aliases," that is, a single variable with two identifiers, perhaps one as a global variable and one as a reference parameter.

21.2.2 Procedure Statements

A procedure statement executes the procedure that the procedure identifier denotes. For example, if you defined the procedure DO_IT:

```
PROCEDURE DO_IT;
BEGIN
  WRITELN('Did it')
END;
```

DO_IT is now a statement that you can execute simply by typing its name:

```
DO_IT
```

If you declare the procedure with a formal parameter list, the procedure statement must include the actual parameters.

Wang PC Pascal includes a large number of predeclared procedures. (See Chapter 23 for complete information on these). One of the predeclared procedures is ASSIGN. You need not declare it in order to use it.

The ASSIGN procedure contains a parameter list. These parameters are the actual parameters bound to the formal parameters in the procedure declaration. For a discussion of formal and reference parameters, see Section 22.4.

21.2.3 The GOTO Statement

A GOTO statement indicates that further processing continues at another part of the program text, namely at the place of the label. You must declare a LABEL in a LABEL declaration section before using it in a GOTO statement.

The following restrictions apply to the use of GOTO statements:

- A GOTO must not jump to a more deeply nested statement, that is, into an IF, CASE, WHILE, REPEAT, FOR, or WITH statement. GOTOs from one branch of an IF or CASE statement to another are legal.
- A GOTO from one procedure or function to a label in the main program or in a higher-level procedure or function is legal. A GOTO can jump out of one of these statements as long as the statement is directly within the body of the procedure or function. However, such a jump generates extra code both at the location of the GOTO and at the location of the label. The GOTO and label must be in the same compiland, since labels, unlike variables, cannot receive the PUBLIC attribute.

The following are examples of GOTO statements, both legal and illegal:

```
PROGRAM LABEL_EXAMPLES;
LABEL 1, 2, 3, 4;
```

```
PROCEDURE ONE;
LABEL 11, 12, 13;
```

```
PROCEDURE IN_ONE;
LABEL 21;
{Outer-level GOTOs cannot jump into 21.}
```

```
BEGIN
  IF TUESDAY THEN GOTO 1
  ELSE GOTO 11;
  {1 and 11 are both legal outer level labels.}
  21: WRITE ('IN_ONE')
END;
```

```

BEGIN {Procedure one}
  IF RAINING THEN GOTO 1 ELSE GOTO 11;
  {That was legal.}
  11: GOTO 21;
  {Illegal. Cannot jump into inner level}
  {procedures.}
END;

PROCEDURE TWO;
BEGIN
  GOTO 11
  {Illegal. Cannot jump into different procedure at same level}
END;

BEGIN {Main level}
  IF SEATTLE
  THEN
    BEGIN BEGIN
      GOTO 2;
      {OK to go to 2 at program level.}

      4: WRITE ('here');
    END END
  ELSE GOTO 4;
  {OK to jump into THEN clause.}
  2: GOTO 3;
  {Illegal. Cannot jump into REPEAT statement.}
  REPEAT
    WHILE MS_BYRON DO
      3: GOTO 2
      {OK to jump out of loops.}
    UNTIL DATE;
  1: GOTO 11;
  {Illegal. Cannot jump into procedure from program.}

END.

```

If the \$GOTO metaccommand is on, the compiler flags every GOTO statement with a warning that "GOTOS are considered harmful." This may be useful either in an educational environment or for finding all GOTOS in a program in order to locate a bug. The J (jumps) column of the listing file contains the following:

- A plus (+) or an asterisk (*) flags a GOTO to a label later in the listing.
- A minus sign (-) or an asterisk (*) marks a GOTO to a label already in the listing.

See Section 26.5 for details about the listing file.

21.2.4 The BREAK, CYCLE, and RETURN Statements

At the extend-level, BREAK, CYCLE, and RETURN statements are legal in addition to the simple statements already described. These statements perform the following functions:

- BREAK leaves the currently executing loop.
- CYCLE leaves the current iteration of a loop and starts the next iteration.
- RETURN leaves the current procedure, function, program, or implementation.

All three statements are functionally equivalent to a GOTO statement.

- A BREAK statement is a GOTO to the first statement after a repetitive statement.
- A CYCLE statement is a GOTO to an implied empty statement after the body of a repetitive statement. This jump starts the next iteration of a loop. In either a WHILE or REPEAT statement, CYCLE performs the Boolean test in the WHILE or UNTIL clause before executing the statement again; in a FOR statement, CYCLE goes to the next value of the control variable.
- A RETURN statement is a GOTO to an implied empty statement after the last statement in the current procedure or function or the body of a program or implementation.

The J (jump) column in the listing file contains a plus sign (+) or an asterisk (*) for a BREAK statement, a minus sign (-) or an asterisk (*) for a CYCLE statement, and an asterisk (*) for a RETURN statement. See Section 26.5 for information about the listing file.

BREAK and CYCLE have two forms, one with a loop label and one without. If you give a loop label, the label identifies the loop to leave or restart. If you do not give a label, the compiler assumes the innermost loop, as the following example shows:

```
OUTER: FOR I := 1 TO N1 DO
  INNER: FOR J := 1 TO N2 DO
    IF A [I, J] = TARGET THEN BREAK OUTER;
```

21.3 STRUCTURED STATEMENTS

Structured statements are themselves composed of other statements. Four kinds of structured statements exist:

Statements

1. Compound statements
2. Conditional statements
3. Repetitive statements
4. WITH statement

The control level is shown in the C (control) column of the listing file. The value in the C column increases by one each time control passes to a nested statement; conversely, this value decreases by one each time control passes back to the nesting statement. This helps you search for a missing or extra END in a program.

21.3.1 Compound Statements

The compound statement is a sequence of simple statements, enclosed by the reserved words BEGIN and END. The components of a compound statement execute in the same sequence as they appear in the source file. The following are examples of compound statements:

```
BEGIN
  TEMP := A [I];
  A[I] := A [J];
  A [J] := TEMP
  {Semicolon not needed here.}
END
```

```
BEGIN
  OPEN_DOOR;
  LET_EM_IN;
  CLOSE_DOOR;
  {Semicolon signifies empty statement.}
END
```

All Wang PC Pascal conditional and repetitive control structures (except REPEAT) operate on a single statement, not on multiple statements with ending delimiters. In this context, BEGIN and END serve as punctuation, like semicolon, colon, or parentheses. If you prefer, you can substitute a pair of square brackets for the BEGIN and END pair of reserved words. A right bracket (]) can only match a left bracket ([) (not a BEGIN, CASE, or RECORD). In other words, a right bracket is not a synonym for END.

You cannot use brackets as synonyms for BEGIN and END to enclose the body of a program, implementation, procedure, or function; you can only use BEGIN and END for this purpose. The following are examples of brackets replacing BEGIN and END:

```

IF FLAG THEN [X := 1; Y := -1]
ELSE [X := -1; Y := 0];

WHILE P.N <> NIL DO
  [Q := P; P := P.N; DISPOSE (Q)];

FUNCTION R2 (R: REAL): REAL;
  [R2 := R * 2]
  [Illegal.]

```

21.3.2 Conditional Statements

A conditional statement selects only one of its component statements for execution. The conditional statements are the IF and CASE statements. Use the IF statement for one or two conditions, the CASE statement for multiple conditions.

The IF Statement

The IF statement allows for conditional execution of a statement. If the Boolean expression following the IF is true, the statement following the THEN executes. If the Boolean expression following the IF is false, the statement following the ELSE, if present, executes. The following are examples of IF statements:

```

IF I > 0 THEN I := I - 1
{No semicolon here.}
ELSE I := I + 1

IF (I <= TOP) AND (ARRI [I] <> TARGET) THEN
  I := I + 1

IF I <= TOP THEN
  IF ARRI [I] <> TARGET THEN
    I := I + 1

IF I = 1 THEN
  IF J = 1 THEN
    WRITELN('I equals J')
  ELSE
    WRITELN('DONE only if I = 1 and J <> 1')
    {This ELSE corresponds to the most deeply}
    {nested IF. Thus, the second WRITELN}
    {executes only if I = 1 and J <> 1.}

```

```

IF I = 1 THEN BEGIN
  IF J = 1 THEN WRITELN('I equals J')
  END
ELSE
  WRITELN('DONE only if I <> 1')
  {Now the ELSE is paired with the first IF,}
  {since the second IF statement is}
  {bracketed by the BEGIN/END pair. Thus,}
  {the second WRITELN is executed if I <> 1.}

```

A semicolon before an ELSE is always incorrect. The compiler skips it during compilation and issues a warning message.

The Boolean expression following an IF can include the sequential control operators described in Section 21.3.5.

The CASE Statement

The CASE statement consists of an expression (called the CASE index) and a list of statements. A constant list, called a CASE constant list, precedes each statement. The only statement that executes is the one whose CASE constant list contains the current value of the CASE index. The CASE index and all constants must be of compatible, ordinal types. The following are examples of CASE statements:

```

CASE OPERATOR OF
  PLUS: X := X + Y;
  MINUS: X := X - Y;
  TIMES: X := X * Y
END
{OPERATOR is the CASE index. PLUS, MINUS, and}
{TIMES are CASE constants. In this instance,}
{they are all of the values assumable by the}
{enumerated variable, OPERATOR.}

CASE NEXTCH OF
  'A'..'Z', ' ' : IDENTIFIER;
  '+', '-', '*', '/' : OPERATOR;
  {Commas separate CASE constants}
  {and ranges of CASE constants.}
  OTHERWISE
    WRITE ('Unknown Character')
    {i.e., if any other character}
END

```

The CASE constant syntax is the same as for RECORD variant declarations. In standard Pascal, a CASE constant is one or more constants separated by commas. At the extend-level, you can substitute a range of constants, such as 'A'..'Z', for a constant. No constant value can apply to more than one statement. The extend-level also allows the CASE statement to end with an OTHERWISE clause. The OTHERWISE clause contains additional statements that execute if the CASE index value is not in the given set of CASE constant values. One of two things happens if the CASE index value is not in the set and no OTHERWISE clause is present:

- If the range-checking switch is on, this generates a runtime error.
- If the range-checking switch is off, unpredictable results will occur.

In Wang PC Pascal, control does not automatically pass to the next executable statement as in UCSD Pascal and some other languages. If you want this effect, include an empty OTHERWISE clause.

A semicolon can appear after the final statement in the list, but is not required. The compiler skips over a colon (:) after an OTHERWISE and issues a warning.

Depending on optimization, the code the compiler generates for a CASE statement can be either a jump table or a series of comparisons (or both). If it is a jump table, a jump to an arbitrary location in memory can occur if the control variable is out of range and the range-checking switch is off.

21.3.3 Repetition Statements

Repetition statements specify repeated execution of a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

The extend-level in Wang PC Pascal has two additional statements, BREAK and CYCLE, for leaving or restarting statements that repetition statements reference. These statements are functionally equivalent to a GOTO, but are easier to use.

The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false. The following are examples of WHILE statements:

```
WHILE P <> NIL DO P := NEXT (P)
```

```
WHILE NOT MICKEY DO
  BEGIN
    NEXTMOUSE;
    MICE := MICE + 1
  END
```

The Boolean expression in a WHILE statement may include the sequential control operators described in Section 21.3.5.

Use WHILE if it is possible that no iterations of the loop may be necessary; use REPEAT where you expect at least one iteration of the loop to occur.

The REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times until a Boolean expression becomes true. The following are examples of REPEAT statements:

```
REPEAT
  READ (LINEBUFF);
  COUNT := COUNT + 1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

The Boolean expression in a REPEAT statement may include the sequential control operators described in Section 21.3.5.

Use the REPEAT statement to execute statements (not just a single statement) one or more times until a condition becomes true. This differs from the WHILE statement, in which a single statement may not execute at all.

The FOR Statement

The FOR statement tells the compiler to execute a statement repeatedly while it assigns a progression of values to the control variable of the FOR statement. The values start with the "initial value" and end with the "final value".

The FOR statement has two forms. In the first, the control variable increases in value. In the second, the control variable decreases in value. An example of each form follows:

```
FOR I := 1 TO 10 DO
  {I is the control variable.}
  SUM := SUM + VICTORVECTOR [I]

FOR CH := 'Z' DOWNT0 'A' DO
  {CH is the control variable.}
  WRITE (CH)
```

You can also use a FOR statement to step through the values of a set, as shown:

```
FOR TINT :=
  LOWER (SHADES) TO UPPER (SHADES) DO
  IF TINT IN SHADES
    THEN PAINT_AREA (TINT);
```

The ISO standard gives explicit rules about the control variable in FOR statements:

- The control variable must be of an ordinal type.
- The control variable must also be an entire variable, not a component of a structure.

- The control variable must be local to the immediately enclosing program, procedure, or function. It cannot be a reference parameter of the procedure or function. However, at the extend-level of Wang PC Pascal, the control variable can also be any STATIC variable, unless the variable has a segmented ORIGIN attribute. Using a program level variable is an ISO error that the compiler does not detect.
- Assignments to the control variable are illegal in the repeated statement. To detect this error, make the control variable READONLY within the FOR statement; the compiler does not detect this error when a procedure or function the repeated statement invokes alters the control variable. You cannot pass the control variable as a VAR (or VARS) parameter to a procedure or function.
- The initial and final values of the control variable must be compatible with the type of the control variable. If the statement executes, both the initial and final values must also be assignment-compatible with the control variable. The compiler evaluates the initial value first, and then the final value. The compiler evaluates both values only once before the statement executes.

The statement following the DO does not execute at all if:

- The initial value is greater than the final value in the TO case.
- The initial value is less than the final value in the DOWNT0 case.

The sequence of values the control variable receives starts with the initial value. The SUCC function for the TO case or the PRED function for the DOWNT0 case defines this sequence until the last execution of the statement, when the control variable has its final value. The value of the control variable after a FOR statement terminates naturally (whether or not the body executes) is mathematically undefined. It can vary due to optimization; \$INITCK can set it to an uninitialized value. The value the control variable has after it leaves a FOR statement with GOTO or BREAK, however, is the value it had at the time of exit.

In standard Pascal, the body of a FOR statement may or may not execute, so a test is necessary to see whether the body should execute at all. If the control variable is of type WORD (or a subrange), however, and its initial value is a constant zero, the body must execute no matter what the final value. In this case, it is not necessary to perform an extra test.

A control variable with the STATIC attribute may be more efficient than one that is not.

At the extend-level in Wang PC Pascal, you can use temporary control variables:

```
FOR VAR <control-variable>
```

Statements

The prefix VAR declares the control variable as local to the FOR statement (that is, at a lower scope). You need not declare the control variable in a VAR section. Such a control variable is not available outside the FOR statement, and any other variable with the same identifier is not available within the FOR statement itself. Other synonymous variables are, however, available to procedures or functions called within the FOR statement. The following are examples of temporary control variables:

```
FOR VAR I := 1 TO 100 DO
    SUM := SUM + VICTOR [I]

FOR VAR COUNTDOWN := 10 DOWNT0 LIFT_OFF DO
    MONITOR_ROCKET
```

The BREAK and CYCLE Statements

In theory, a program that uses the Wang PC Pascal extend-level BREAK and CYCLE statements does not need to use any GOTO statements. Each of these two statements has two forms, one with a loop label and one without. A loop label is a normal GOTO label prefixed to a FOR, WHILE, or REPEAT statement. Because you can use identifier labels at the extend level, a suggested practice is to use integers for labels GOTOs reference and identifiers for loop labels. The following are examples of CYCLE and BREAK statements:

```
LABEL SEARCH, CLIMB;
.
.
SEARCH: WHILE I <= I_TOP DO
    IF PILE [I] = TARGET THEN BREAK SEARCH
    ELSE I := I + 1;
.
.
FOR I := 1 TO N DO
    IF NEXT [I] = NIL THEN BREAK;
.
.
CLIMB: WHILE NOT ITEM↑.LEAF DO
    BEGIN
        IF ITEM↑.LEFT <> NIL
            THEN [ITEM := ITEM↑.LEFT; CYCLE CLIMB];
        IF ITEM↑.RIGHT <> NIL
            THEN [ITEM := ITEM↑.RIGHT; CYCLE CLIMB];
        WRITELN ('Very strange node');
        BREAK CLIMB
    END;
```

21.3.4 The WITH Statement

The WITH statement opens the scope of a statement to include the fields of one or more records, so you can refer to the fields directly. For example, the following statements are equivalent:

```
WITH PERSON DO WRITE (NAME, ADDRESS, PHONE)
WRITE (PERSON.NAME, PERSON.ADDRESS, PERSON.PHONE)
```

The record in a WITH statement can be a variable, constant identifier, structured constant, or function identifier; it cannot be a component of a PACKED structure. If you use a function identifier, it refers to the function's local result variable. If the record in a WITH statement is a file buffer variable, the compiler issues a warning, as changing the position in the WITH statement may cause an error.

The record can also be any expression in parentheses, in which case the compiler evaluates the expression and assigns the result to a temporary (hidden) variable. If you want to evaluate a function designator, you must enclose it in parentheses.

You can give a list of records after the WITH, separated by commas. Each record so listed must be of a different type from all the others, because the field identifiers refer only to the last instance of the record with the type. These statements are equivalent:

```
WITH PMODE, QMODE DO statement
WITH PMODE DO WITH QMODE DO statement
```

The compiler selects any record variable of a WITH statement that is a component of another variable before the statement executes. Do not pass active WITH variables as VAR or VARS parameters or pass their pointers to the DISPOSE procedure. However, the compiler does not catch these errors. Assignments to any of the record variables in the WITH list or components of these variables are legal if the WITH record is a variable.

In Wang PC Pascal, every WITH statement allocates an address variable that holds the address of the record. If the record variable is on the heap, do not dispose the pointer to it within the WITH statement. If the record variable is a file buffer, do not perform any I/O to the file within the WITH statement.

21.3.5 Sequential Control

To increase execution speed or to guarantee correct evaluation, it is often useful in IF, WHILE, and REPEAT statements to treat the Boolean expression as a series of tests. If one test fails, the remaining tests do not execute. Two extend-level operators in Wang PC Pascal provide for such tests:

Statements

- AND THEN -- X AND THEN Y is false if X is false; AND THEN evaluates Y only if X is true.
- OR ELSE -- X OR ELSE Y is true if X is true; OR ELSE evaluates Y only if X is false.

If you use several sequential control operators, the compiler evaluates them strictly from left to right.

You can include only these operators in the Boolean expression of an IF, WHILE, or UNTIL clause; you cannot use them in other Boolean expressions. Furthermore, they cannot occur in parentheses, and the compiler evaluates them after all other operators. The following are examples of sequential control operators:

```

IF SYM <> NIL AND THEN SYM↑.VAL < 0 THEN
  NEXT_SYMBOL

WHILE I <= MAX AND THEN VECT [I] <> KEY DO
  I := I + 1;

REPEAT GEN (VAL)
UNTIL VAL = 0 OR ELSE (QU DIV VAL) = 0;

WHILE POOR AND THEN GETTING_POORER
  OR ELSE BROKE AND THEN BANKRUPT DO
  GET_RICH

```


CHAPTER 22

INTRODUCTION TO PROCEDURES AND FUNCTIONS

Procedures and functions act as subprograms that execute under the supervision of a main program. Unlike programs, however, you can nest procedures and functions within each other; they can even call themselves. Furthermore, they have sophisticated parameter passing capabilities that programs lack.

You invoke procedures as program statements; you can invoke functions in program statements wherever a value is needed.

The general format for procedures and functions is similar to the format for programs. The three-part structure includes a heading, declarations, and a body. The following is an example of a procedure declaration:

```
{Heading}
PROCEDURE MODEL (I: INTEGER; R: REAL);

{Beginning of declaration section}
LABEL 123;
CONST ATOP = 199;
TYPE INDEX = 0..ATOP;
VAR ARAY: ARRAY [INDEX] OF REAL; J: INDEX;

{Function declaration}
FUNCTION FONE (RX: REAL): REAL;
BEGIN
    FONE := RX * I
END;

{Procedure declaration}
PROCEDURE FOUT (RY: REAL);
BEGIN
    WRITE ('Output is ', RY)
END;
```

```

(Body of procedure MODEL)
BEGIN
  FOR J := 0 TO ATOP DO
    IF GLOBALVAR THEN
      {Activation of procedure FOUT with}
      {value returned by function FONE.}
      FOUT (FONE (R + ARAY [J]))
    ELSE GOTO 123;
  123: WRITELN ('Done');
END;

```

The declarations and body together are the block.

The declaration of a procedure or function associates an identifier with a portion of a program. Later, you can activate that portion of the program with the appropriate procedure statement or function designator.

22.1 PROCEDURES

The previous example illustrates the general format of a procedure declaration. The heading is followed by:

- Declarations for labels, constants, types, variables, and values
- Local procedures and functions
- The body, which is enclosed by the reserved words BEGIN and END

When the body of a procedure finishes execution, control returns to the program element that called it.

At the standard level, the order of declarations must be as follows:

1. LABEL
2. CONST
3. TYPE
4. VAR
5. Procedures and functions

At the extend-level, you can have any number of LABEL, CONST, TYPE, VAR, and VALUE sections, as well as procedure and function declarations, in any order. Although you can mix data declarations (CONST, TYPE, VAR, VALUE) with procedure and function declarations, it is usually clearer to give all data declarations first.

Putting variable declarations after procedure and function declarations guarantees that none of the procedures or functions use these variables.

Introduction to Procedures and Functions

In general, the initial values of variables are undefined. The **VALUE** section, which should follow the **VAR** section, is a Wang PC Pascal extension that lets you explicitly initialize program, module, implementation, **STATIC**, and **PUBLIC** variables. If the initialization switch (**\$INITCK**) is on, it sets all **INTEGER**, **INTEGER** subrange, **REAL**, and pointer variables to an uninitialized value. File variables are already initialized, regardless of the setting of the initialization switch.

22.2 FUNCTIONS

Functions are the same as procedures, except that you invoke them in an expression instead of a statement and they return a value.

Function declarations define the parts of a program that compute a value. Evaluating a function designator, which is part of an expression, automatically evaluates a function.

A function declaration has the same format as a procedure declaration, except that the heading also gives the type of value the function returns. The following is an example of a function heading:

```
FUNCTION MAXIMUM (I, J: INTEGER): INTEGER;
```

Within the block of a function, either in the body itself or in a procedure or function nested within the block, at least one assignment to the function identifier must execute to set the return value. The compiler does not check for this assignment at runtime unless the initialization switch is on and the returned type is **INTEGER**, **REAL**, or a pointer. If there is no assignment at all to the function identifier, however, the compiler issues an error message.

At the standard level, functions can return any simple type (ordinal, **REAL**, or **INTEGER**) or a pointer. At the extend-level, functions can return any simple, structured, or reference type. They cannot return any type that cannot be assigned (that is, a super array type or a structure that contains a file, although a super array derived-type is legal).

A function identifier in an expression invokes the function recursively rather than giving the current value of the function.

To obtain the current value of a function, use the **RESULT** function, which takes the function identifier as a parameter and is available at the extend-level. The following is an example of **RESULT** function used to obtain the current value of a function within an expression:

```
FUNCTION FACT (F: REAL): REAL;
BEGIN
  FACT := 1;
  WHILE F > 1 DO
    BEGIN
      FACT := RESULT (FACT) * F; F := F-1
    END
  END
```

Using the RESULT function is more efficient than using a separate local variable for the value of the function and then assigning this local variable to the function identifier before returning. If the function has a structured value, the usual component selection syntax can follow the RESULT function.

A function identifier on the left side of an assignment refers to the function's local variable, which contains its current value, instead of invoking the function recursively. Other places where using the function identifier refers to this local variable are the following:

- A reference parameter
- The record of a WITH statement
- The operand of an ADR or ADS operator

11 of these uses involve getting the address (not the value) of a variable.

Instead of using the function's local variable, you may want to invoke the function and use the return value. As mentioned in Section 19.1, getting the address of an expression involves evaluating the expression, putting the resulting value into a temporary (hidden) variable, and using the address of this variable.

To do this for a function, you must force evaluation by putting the function designator in parentheses, as shown:

```
TYPE IREC = RECORD I: INTEGER END;

FUNCTION SUM (A, B: INTEGER): IREC;
{Return sum of A and B.}
BEGIN
  IF TUESDAY THEN
    BEGIN
      IF B = 0 THEN BEGIN SUM := A; RETURN END;
      WITH (SUM (A,B-1)) {Call SUM recursively.}
      DO SUM.I := I + 1 {I is result of call.}
    END
  ELSE
    BEGIN
      WITH SUM {Use function's}
      DO I := A + B; {local variable.}
    END
  END;
END;
```

22.3 ATTRIBUTES AND DIRECTIVES

An attribute gives additional information about a procedure or function. Attributes are available at the extend-level of Wang PC Pascal. They follow the heading, enclosed in brackets and separated by commas. Available attributes include ORIGIN, PUBLIC, FORTRAN, PURE, and INTERRUPT.

Introduction to Procedures and Functions

A directive gives information about a procedure or function, but it also indicates that only the heading of the procedure or function occurs, by replacing the block (declarations and body) normally included after the heading. Directives are available in standard Pascal. **EXTERN** and **FORWARD** are the only directives available. You can use **EXTERN** only with procedures or functions directly nested in a program, module, implementation, or interface. This restriction prevents access to nonlocal stack variables.

Table 22-1 displays the attributes and directives that apply to procedures and functions. The following subsections describe these attributes in detail.

Table 22-1. Attributes and Directives for Procedures and Functions

Name	Purpose
FORWARD	A directive. Lets you call a procedure or function before you give its block in the source file.
EXTERN	A directive. Indicates that a procedure or function resides in another loaded module.
PUBLIC	An attribute. Indicates that a procedure or function may be accessed by other loaded modules.
ORIGIN	An attribute. Tells the compiler where the code for an EXTERN procedure or function resides.
FORTRAN	An attribute. Specifies a calling sequence for compatibility with Wang PC FORTRAN.
INTERRUPT	An attribute. Gives a procedure a special calling sequence that saves program status on the stack.
PURE	An attribute. Signifies that the function does not modify any global variables.

The following rules apply when you combine attributes in the declaration of procedures and functions:

- You can give any function the PURE attribute.
- You must nest procedures and functions with attributes directly within a program, module, or unit. The only exception to this rule is the PURE attribute.
- A given procedure or function can have only one calling sequence attribute (either FORTRAN or INTERRUPT, but not both).
- PUBLIC and EXTERN are mutually exclusive, as are PUBLIC and ORIGIN.

The compiler gives the EXTERN or FORWARD directive automatically to all constituents of the interface of a unit; in the implementation, the compiler gives PUBLIC automatically to all constituents that are not EXTERN.

Because you declare the constituents of a unit only in the interface (not in the implementation), the interface is where you give the attributes. You can give the EXTERN directive in an implementation by declaring all EXTERN procedures and functions first; you cannot use ORIGIN in either the interface or implementation of a unit.

In a module, you can give a group of attributes in the heading to apply to all directly nested procedures and functions. The only exception to this rule is the ORIGIN attribute, which can apply only to a single procedure or function.

If the PUBLIC attribute is one of a group of attributes in the heading of a module, an EXTERN attribute within the module explicitly overrides the global PUBLIC attribute. If the module heading has no attribute clause, the compiler assumes the PUBLIC attribute for all directly nested procedures and functions.

The PUBLIC attribute allows other loaded code to call a procedure or function. You cannot use PUBLIC with the EXTERN directive. The EXTERN directive permits a call to some other loaded code, using either the ORIGIN address or the Linker. PUBLIC, EXTERN, and ORIGIN provide a low-level way to link Wang PC Pascal routines with other routines in Wang PC Pascal or other languages.

A procedure or function declaration with the EXTERN or FORWARD directive consists only of the heading, without an enclosed block. EXTERN routines have an implied block outside of the program. FORWARD routines have a block later in the same compiland. Both directives are available at the standard level of Wang PC Pascal. The keyword EXTERNAL is a synonym for EXTERN.

The PURE attribute applies only to functions, not to procedures. Conversely, INTERRUPT applies only to procedures, not to functions. PURE is the only attribute that you can use in nested functions.

Introduction to Procedures and Functions

22.3.1 The FORWARD Directive

A FORWARD declaration allows you to call a procedure or function before you fully declare it in the source text. This permits indirect recursion, where A calls B and B calls A. You make a FORWARD declaration by specifying a procedure or function heading, followed by the directive FORWARD. Later, you actually declare the procedure or function, without repeating the formal parameter list or any attributes. The following is an example of a FORWARD declaration:

```
{Declaration of ALPHA, with parameter}
{list and attributes}
FUNCTION ALPHA (Q, R: REAL): REAL [PUBLIC];
FORWARD;
```

```
{Call for ALPHA}
PROCEDURE BETA (VAR S, T: REAL);
BEGIN
  T := ALPHA (S, 3.14)
END;
```

```
{Actual declaration of ALPHA,}
{without parameter list}
FUNCTION ALPHA;
BEGIN
  ALPHA := (Q + R);
  IF R < 0.0 THEN BETA (3.14, ALPHA);
END;
```

22.3.2 The EXTERN Directive

The EXTERN directive identifies a procedure or function that resides in another loaded module. You give only the heading of the procedure or function, followed by the word EXTERN. The compiler presumes that the actual implementation of the procedure or function exists in some other module.

EXTERN is an attribute when used with a variable, but a directive when used with a procedure or function. As with variables, the keyword EXTERNAL is a synonym for EXTERN.

The EXTERN directive for a particular procedure or function within a module overrides the PUBLIC attribute for the entire module. The EXTERN directive is also legal in an implementation of a unit for a constituent procedure or function. You must declare all such external constituents at the beginning of the implementation, before all other procedures and functions.

You must nest any procedure or function with the EXTERN directive directly within a program. You can also link Wang PC Pascal routines by linking separately compiled units (see Chapter 25). The following are examples of procedure and function headings declared with the EXTERN directive:

```

FUNCTION POWER (X, Y: REAL): REAL; EXTERN;

PROCEDURE ACCESS (KEY: KTYP) [ORIGIN SYSB+4];
EXTERN;

```

These examples declare the function POWER and the procedure ACCESS EXTERN. External compilands implement both of these functions. ACCESS also has the ORIGIN attribute, which is discussed in Section 22.3.4.

You cannot declare a procedure or function EXTERN if you have previously declared it FORWARD.

22.3.3 The PUBLIC Attribute

The PUBLIC attribute indicates a procedure or function that you can access from other loaded modules. In general, you access PUBLIC procedures and functions from other loaded modules by declaring them EXTERN in the modules that call them. Thus, you declare a procedure PUBLIC and define it in one module, and use it in another simply by declaring it EXTERN in another module.

As with variables, the compiler passes the identifier of the procedure or function to the Linker, where the Linker can truncate it as necessary. See Appendix A for specific information on limitations the compiler and Linker can set on identifiers. PUBLIC and ORIGIN are mutually exclusive; PUBLIC routines need a following block, and ORIGIN routines must be EXTERN.

You must nest any procedure or function with the PUBLIC attribute directly within a program or implementation. A higher-level way to link Wang PC Pascal routines is by linking separately compiled units. See Chapter 25 for details. The following are examples of procedures and functions declared PUBLIC:

```

FUNCTION POWER (X, Y: REAL): REAL [PUBLIC];
{The function POWER is available to other modules}
{because it was declared PUBLIC.}
BEGIN
.
.
END;

PROCEDURE ACCESS (KEY: KTYP)
[ORIGIN SYSB+4, PUBLIC];
BEGIN
.
.
END;
{Illegal since ORIGIN must also be EXTERN.}

```

22.3.4 The ORIGIN Attribute

You must use the ORIGIN attribute with the EXTERN directive; ORIGIN tells the compiler where it can find the procedure or function directly, so the Linker does not require a corresponding PUBLIC identifier. The following are examples of procedures and functions given the ORIGIN attribute:

```
PROCEDURE OPSYS [ORIGIN 8, FORTRAN];
EXTERN;

FUNCTION A_TO_D (C: SINT): SINT [ORIGIN #100];
EXTERN;
```

In the first example, the procedure OPSYS begins at the absolute decimal address 8, has the FORTRAN calling sequence (described in Section 22.3.5), and is declared EXTERN. In the second example, the function A_TO_D takes a SINT value as a parameter (SINT is the predeclared integer subrange from -127 to +127). The function is located at the hexadecimal address 100.

As with ORIGIN variables, the compiler uses the address to find the code and gives no directives to the Linker. This permits, for example, calling routines at fixed addresses in ROM. In simple cases, it can substitute for a linking loader.

Remember that ORIGIN always implies EXTERN. Thus, you cannot declare procedures or functions with the ORIGIN attribute that you previously declared FORWARD. In addition, you cannot give ORIGIN as an attribute after the module heading.

Currently, you cannot use the ORIGIN attribute with a constituent of a unit, either in an interface or in an implementation.

As with variables, the origin can be a segmented address. On segmented machines, a nonsegmented procedural origin assumes the current code segment with the offset given with the attribute; this form has no obvious uses.

22.3.5 The FORTRAN Attribute

The FORTRAN attribute applies both to procedures and functions (but not to variables). Instead of the usual Pascal calling sequence, it specifies a calling sequence that is compatible with the Wang PC FORTRAN compiler on your machine. This lets you call a Wang PC Pascal procedure or function from Wang PC FORTRAN programs and, conversely, external FORTRAN subroutines or Wang PC FORTRAN functions from a Wang PC Pascal program. The following is an example of a procedure with the FORTRAN attribute:

```
PROCEDURE DELTA (I, J: INTEGER) [FORTRAN];
FORWARD;
```

You must nest any procedure or function with the FORTRAN attribute directly within a program or implementation.

In a 16-bit environment, Wang PC Pascal uses the same calling sequence as the Wang PC FORTRAN, BASIC, and COBOL compilers. Thus, there is no need to give the FORTRAN attribute; if you do, the Wang PC Pascal Compiler ignores it. See Appendix A for details on the Wang PC Pascal calling sequence.

22.3.6 The INTERRUPT Attribute

The INTERRUPT attribute applies only to procedures (not to functions or variables). It gives a procedure a special calling sequence that saves program status on the stack, which in turn allows the compiler to process a hardware interrupt, restore status, and return control to the program, all without affecting the current state of the program. The following is an example of a procedure with the INTERRUPT attribute:

```
PROCEDURE INCHAR [INTERRUPT];
```

Because hardware interrupts invoke procedures with the INTERRUPT attribute, you cannot invoke them with a procedure statement. You can only invoke an INTERRUPT procedure when the interrupt associated with it occurs. Also, INTERRUPT procedures take no parameters.

Declaring a procedure with the INTERRUPT attribute ensures that the procedure conforms to the constraints of an interrupt handler in which:

- A special calling sequence saves all status on the stack
- The status the sequence saves includes machine registers and flags, plus any special global compiler data such as the frame pointer
- The status the sequence saves is restored when it leaves the procedure

You must nest all INTERRUPT procedures directly within a compiland.

Interrupts are not automatically vectored to INTERRUPT procedures; further, insofar as possible on the target machine, an INTERRUPT procedure neither enables nor disables interrupts. Interrupt vectoring and enabling are too machine-dependent to be included in a machine-independent language such as Wang PC Pascal.

Wang PC Pascal does, however, provide the VECTIN library procedure, which takes an interrupt level and an interrupt procedure as parameters and sets the interrupt vector in a machine-dependent way. Similarly, the library procedures ENABIN and DISBIN, respectively, enable and disable interrupts in a machine-dependent way. See Chapter 23 for more information on these routines. See also Appendix A for information about the implementation of these routines under your operating system.

An INTERRUPT procedure should usually return normally, in order to continue processing in the interrupted routine. This means the following:

- You should not execute a GOTO that leaves an INTERRUPT procedure.

Introduction to Procedures and Functions

- You should turn all debug checking off (\$DEBUG-, \$ENTRY-, and \$RUNTIME-).
- The compiler may not check stack overflow even if \$STACKCK is on.

INTERRUPT procedures introduce reentry into Wang PC Pascal code.

Semaphores protect some critical sections in the runtime system. These semaphores generate a runtime error if a critical section is locked. For example, if the heap allocator is executing when an interrupt occurs and the INTERRUPT procedure tries to allocate a block from the heap, the structure of the heap could become invalid. This condition causes a runtime error.

In most cases, however, a semaphore does not protect the file system. Therefore, it is safest to avoid performing any I/O within the INTERRUPT procedure. Alternatively, you can avoid most problems with I/O in an INTERRUPT procedure by not opening or closing any files (not declaring any local file variables or creating files on the heap) and by not performing input or output with any file that might be in the process of performing I/O when the interrupt occurs.

22.3.7 The PURE Attribute

The PURE attribute applies only to functions, not to procedures or variables. PURE indicates to the compiler's optimizer that the function does not modify any global variables either directly or by calling some other procedure or function. The following is an example of a PURE declaration:

```
FUNCTION AVERAGE (CONST TABLE: RVECTOR):  
REAL [PURE];
```

For further illustration, examine these statements:

```
A := VEC [I * 10 + 7];  
B := FOO;  
C := VEC [I * 10 + 9]
```

If the function FOO has the PURE attribute, the optimizer only generates code to compute I*10 once. However, FOO, if it is not PURE, can modify I so that the compiler must recompute I*10 after the call to FOO.

Functions are not considered PURE unless they have the attribute explicitly. The compiler checks to see that a PURE function does not do any of the following:

- Assign to a nonlocal variable
- Have any VAR or VARS parameters (CONST and CONST parameters are legal)
- Call any functions that are not PURE

A PURE function also should not:

- Use the value of a global variable
- Modify the referents of references (pointer or address type referents) values pass
- Do input or output

Since the result of a PURE function with the same parameters must always be the same, the optimizer may optimize the entire function call away. For example, if DSIN is PURE in the following statements, the compiler only calls DSIN once:

```
HX := A * DSIN (P[I, J] * 2);
HY := B * DSIN (P[I, J] * 2);
```

22.4 PROCEDURE AND FUNCTION PARAMETERS

Procedures and functions can take three different type of parameters:

- Value parameters
- Reference parameters
- Procedural and functional parameters

Each of these is discussed separately, in the order listed, in the following subsections.

The discussion mentions both formal and actual parameters. A formal parameter is the parameter you give when you declare the procedure or function with an identifier in the heading. When the program calls the function or procedure, an actual parameter substitutes for the formal parameter given earlier; here the parameter takes the form of a variable, value, or expression.

Extend-level Wang PC Pascal can perform the following parameter features:

- Pass a super array type as a reference parameter.
- Declare a reference parameter READONLY.
- Declare explicit segmented reference parameters.

Introduction to Procedures and Functions

22.4.1 Value Parameters

When a program passes a value parameter, the actual parameter is an expression. The compiler evaluates that expression in the scope of the calling procedure or function and assigns it to the formal parameter. The formal parameter is a variable local to the procedure or function called. Thus, formal value parameters are always local to a procedure or function. The following is an example of value parameters:

```
{Function declaration }
FUNCTION ADD (A, B, C : REAL): REAL;
  {A, B, and C are formal parameters }
```

```
X := ADD (Y, ADD (1.111, 2.222, 3.333), (Z * 4) )
```

In this particular function invocation, Y, ADD(...), and (Z * 4) are the expressions that make up the actual parameters. In this example, these expressions must all evaluate to the type REAL. (The example also recursively calls the function ADD.)

The actual parameter expression must be assignment-compatible with the type of the formal parameter.

Passing structured types by value is legal; however, it is inefficient, since the compiler must copy the entire structure. A value parameter of a SET, LSTRING, or subrange type can also require a runtime error check if the range-checking switch is on. In addition, SET and LSTRING value parameters can require extra generated code for size adjustment.

You cannot pass a file variable or super array variable as a value parameter, because it cannot be assigned. However, you can pass a variable with a type derived from a super array or file buffer variable. Passing a file buffer variable as a value parameter implies normal evaluation of the buffer variable.

22.4.2 Reference Parameters

When you pass a reference parameter at the standard level of Wang PC Pascal, the keyword VAR precedes the formal parameter. Furthermore, the actual parameter must be a variable, not an expression. The formal parameter denotes this actual variable during the execution of the procedure. Any operation on the formal parameter occurs immediately on the actual parameter, by passing the machine address of the actual variable to the procedure. For target processors with segmentation support, this address is an offset into the default data segment. The following is an example of variable parameters:

```

PROCEDURE CHANGE_VARS (VAR A, B, C : INTEGER);
{A, B, and C are formal reference parameters.}
{They denote variables, not values.}

```

```

CHANGE_VARS (X, Y, Z);

```

In this example, X, Y, and Z must be variables, not expressions. Also, the variables X, Y, and Z change whenever you change the formal parameters A, B, and C in the declared procedure. This differs from the handling of value parameters, which can affect only the copies of values of variables. If the selection of the variable involves indexing an array or dereferencing a pointer or address, these actions execute before the procedure itself. The type of the actual parameter must be identical to the type of the formal parameter.

Passing a nonlocal variable as a VAR parameter puts a slash (/) or percent sign (%) in the G (global) column of the listing file (see Section 26.5 for information about significance of these characters in the G column of the listing)..

You cannot pass the following as VAR parameters:

- A component of a PACKED structure (except CHAR of a STRING or LSTRING)
- Any variable with a READONLY or PORT attribute (includes CONST and CONSTS parameters and the FOR control variable)

Passing a file buffer variable by reference generates a warning message, because it bypasses the normal file system call a buffer variable generates. These calls do not occur when a you pass a file variable by reference.

On a segmented machine, a VAR parameter passes an address that is really an offset into a default data segment. Some cases require access to objects residing in other segments. To pass these objects by reference, you must tell the compiler to use a segmented address that contains both segment register and offset values. The extend-level includes the parameter prefix VARS instead of VAR:

```

PROCEDURE CONCATS (VARS T, S: STRING);

```

You can use VARS as a data parameter only in procedures and functions, not in the declaration section of programs, procedures, and functions. VARS and CONSTS parameters chiefly maintain compatibility with machines that have two different sizes of address spaces. These parameters are not necessary for a machine with a single size address space. On such machines, the reserved words VARS and CONSTS are equivalent to VAR and CONST.

Super Array Parameters

Super array parameters can appear as formal reference parameters. This allows a procedure or function to operate on an array with a particular super array type (also a component type and index type), but without any fixed upper bounds. The formal parameter is a reference parameter of the super array type itself.

The actual parameter type must be a type derived from the super array type or the super array type itself (that is, another reference parameter or dereferenced pointer). Except for comparing LSTRINGs, you cannot assign or compare entire super array type parameters.

The actual upper and lower bounds of the array are available with the UPPER and LOWER functions; this permits routines that can operate on arrays of any size. You can pass an LSTRING actual parameter to a reference parameter of the super array type STRING. Therefore, you can use the super array parameter STRING for procedures and functions that operate on strings of both STRING and LSTRING types. The following are examples of super array parameters:

```
TYPE REALS = ARRAY [0..*] OF REAL;

PROCEDURE SUMRS (VAR X: REALS; CONST X: REALS);
BEGIN
.
END;
```

For more information, refer to Section 15.2.

Constant and Segment Parameters

At the extend-level, a formal parameter following the reserved word CONST implies that the actual parameter is a READONLY reference parameter. This is especially useful for parameters of structured types, which can be constants, since it eliminates the need for a time-consuming value parameter copy. The actual parameter can be a variable, function result, or constant value.

You cannot make assignments to the CONST parameter or to any of its components. CONST super array types are legal. You cannot pass a CONST parameter in one procedure as a VAR parameter to another procedure. However, it is permissible to pass a VAR parameter in one procedure as a CONST parameter in another. The following is an example of a CONST parameter:

```
PROCEDURE ERROR (CONST ERRMSG: STRING);
```

On a segmented machine, a CONST parameter passes an address that is an offset into a default data segment. In some cases, access to objects in other segments is necessary. To pass these objects by reference, you must tell the compiler to use a segmented address that contains both segment register and offset values. The extend-level includes the parameter prefix CONSTS, instead of CONST. Use of CONSTS parameters parallels use of VARS for formal reference parameters. The following is an example of a CONSTS parameter:

Introduction to Procedures and Functions

```
PROCEDURE CAT (VARS T: STRING; CONSTS S: STRING);
```

You can use a CONSTS parameter as a data parameter only in procedures and functions, not in the declaration section of programs, procedures, and functions.

You can also pass the value of an expression as a CONST or CONSTS parameter. The compiler evaluates the expression and assigns it to a temporary (hidden) variable in the frame of the calling procedure or function. You should enclose such an expression in parentheses to force its evaluation.

You can pass a function identifier by reference as a VAR, VARS, CONST, or CONSTS parameter. This also passes the function's local variable, so the call must occur in the function's body or in a procedure or function the function declares.

You can also pass the value a function designator returns, like any expression, as a CONST or CONSTS parameter. Like any expression passed by reference, you must enclose the function designator in parentheses, as shown:

```
PROCEDURE WRITE_ANSWER (CONSTS A: INTEGER)
BEGIN
  Writeln ('THE ANSWER IS ', A)
END;

FUNCTION ANSWER: INTEGER;
BEGIN
  ANSWER := 42;
  WRITE_ANSWER (ANSWER);
  {Pass reference to local variable.}
END;

PROCEDURE HITCH_HIKE;
BEGIN
  WRITE_ANSWER ((ANSWER))
  {Call ANSWER, assign to temporary variable.}
  {pass reference to temporary variable.}
END;
```

22.4.3 Procedural and Functional Parameters

You can use procedural parameters in numerical analysis, in calling some library routines, and in special applications. In numerical analysis, you might pass a function to a procedure or function that finds an integer between limits, a maximum or minimum value, and so on. Some interesting algorithms in areas such as parsing and artificial intelligence also use procedural parameters.

Introduction to Procedures and Functions

When you pass a procedural or functional parameter, the actual identifier is that for a procedure or function. The formal parameter is a procedure or function heading, including any attributes, preceded by the reserved word PROCEDURE or FUNCTION. For example, examine these declarations:

```

TYPE DOOR = (FRONT, BARN, CELL, DOG_HOUSE);
      SPEED = (FAST, SLOW, NORMAL);
      DIRECTION = (OPEN, SHUT);

PROCEDURE OPEN_DOOR_WIDE
  (VAR A : DOOR; B : SPEED; C : DIRECTION);
.
PROCEDURE SLAM_DOOR
  (VAR DR : DOOR; SP : SPEED; DIR : DIRECTION);
.
PROCEDURE LEAVE_AJAR
  (VAR DD : DOOR; SS : SPEED; DD : DIRECTION);

```

All of the procedures in the example have parameter lists of equal length. The types of the parameters are not only compatible, but also identical. You need not give the formal parameters the same name.

A procedural or functional parameter can accept one of these procedures if the procedure or function appears as shown:

```

FUNCTION DOOR_STATUS (PROCEDURE MOVE_DOOR
  (VAR X: DOOR; Y: SPEED; Z: DIRECTION);
  VAR XX: DOOR; YY: SPEED; ZZ: DIRECTION):INTEGER;
  ("PROCEDURE MOVE_DOOR" is the formal procedural)
  {parameter; next two lines are other formal}
  {parameters.}

BEGIN {door_status}
  DOOR_STATUS := 0;
  MOVE_DOOR(XX, YY, ZZ);
  {One of the three procedures declared}
  {previously executes here.}

  IF XX = BARN AND ZZ = SHUT
  THEN DOOR_STATUS := 1;

  IF XX = CELL AND ZZ = OPEN
  THEN DOOR_STATUS := 2

  IF XX = DOG_HOUSE AND ZZ = SHUT
  THEN DOOR_STATUS := 3
END;

```

Use of the procedural parameter MOVEDOOR might occur in program statements as follows:

```

IF DOOR_STATUS
  (SLAM_DOOR, CELL, FAST, SHUT) = 0
THEN
  SOCIETY := SAFE;
IF DOOR_STATUS
  (OPEN_DOOR_WIDE, BARN, SLOW, OPEN) = 0
THEN
  COWS_ARE_OUT := TRUE;
IF DOOR_STATUS
  (LEAVE_AJAR, DOG_HOUSE, SLOW, OPEN) = 0
THEN
  DOG_CAN_GET_IN := TRUE;

```

In each case above, the actual procedure list is compatible with the formal list, both in number and in type of parameters. If the parameter you pass is a functional parameter, the function return value would also have to be of an identical type. In addition, the set of attributes for both the formal and actual procedural type must be the same, except that the compiler ignores the PUBLIC and ORIGIN attributes and EXTERN directive.

You can pass a PUBLIC or EXTERN procedure, or any local procedure at any nesting level, to the same type of formal parameter. However, the PURE attribute and any calling sequence attributes must match. Also, in systems with segmented code addresses, a procedure or function you pass as a parameter to an EXTERN procedure or function must itself be PUBLIC or EXTERN.

In Wang PC Pascal, you can only pass predeclared procedures and functions in called subroutines. Also, the compiler translates READ, WRITE, ENCODE, and DECODE families into other calls based on the argument types, and so you cannot pass them. You can, however, pass corresponding routines in the file unit or encode/decode unit. For example, a READ of an INTEGER becomes a call to RTIFQQ and you can pass this procedure as a parameter.

You cannot pass the following intrinsic procedures and functions as procedure or function parameters:

1. At the standard level of Wang PC Pascal:

ABS	EOLN	PACK	SQR
ARCTAN	EXP	PAGE	SQRT
CHR	LN	PRED	SUCC
COS	NEW	READ	UNPACK
DISPOSE	ODD	READLN	WRITE
EOF	ORD	SIN	Writeln

2. at the extend-level and system level of Wang PC Pascal:

BYLONG	FLOAT4	READFN	SIZEOF
BYWORD	HIBYTE	READSET	TRUNC
DECODE	HIWORD	RESULT	TRUNC4
ENCODE	LOBYTE	RETYPE	UPPER
EVAL	LOWER	ROUND	WRD
FLOAT	LOWORD	ROUND4	

When you activate a procedure or function that you pass as a parameter, any nonlocal variables it accesses are those in effect at the time you pass the procedure or function as a parameter, rather than those in effect when you activate it. Internally, this passes both the address of the routine and the address of the upper frame (in the stack). The following is an example of formal procedure use:

```

PROCEDURE ALPHA;
  VAR I: INTEGER;

  PROCEDURE DELTA;
  BEGIN
    WRITELN('Delta done')
  END;

  PROCEDURE BETA (PROCEDURE XPR);
  VAR GLOB: INTEGER;

  PROCEDURE GAMMA;
  BEGIN GLOB := GLOB + 1 END;

  BEGIN {Start BETA}
    GLOB := 0;
    IF I = 0
    THEN BEGIN
      I := 1; XPR; BETA (GAMMA)
    END
    ELSE BEGIN
      GLOB := GLOB + 1; XPR
    END
  END;

  BEGIN {Start ALPHA}
    I := 0;
    BETA (DELTA)
  END;

```

In this example, formal procedure use:

1. Calls ALPHA.
2. Calls BETA, passing the procedure DELTA.
3. Creates an instance of GLOB on the stack (call it GLOB1).
4. Clears GLOB1 by setting it to zero. Then, since I is 0, it executes the THEN clause, which sets I to one and executes XPR, which it binds to DELTA.
5. Writes 'Delta done' to OUTPUT.
6. Calls BETA recursively. Passes BETA as GAMMA, and, at this time, passes the access path to any nonlocal variables GAMMA uses (such as GLOB1).

7. Creates another instance of GLOB (GLOB2). Clears GLOB2. Increases GLOB2 by one.
8. Calls XPR, which it binds to GAMMA. GAMMA executes and the instance of GLOB active when you passed GAMMA to BETA, GLOB1, increases by one.
9. GAMMA returns, the second BETA call returns, the first BETA call returns, and finally, ALPHA returns.

CHAPTER 23

AVAILABLE PROCEDURES AND FUNCTIONS

All versions of Pascal predeclare many common procedures and functions. You do not have to declare these procedures and functions in a program. Because they are defined in a scope "outside" the program, you can redefine these identifiers.

To promote portability, Wang PC Pascal makes some of the predeclared procedures and functions available only at the extend or at the system level. Wang PC Pascal also includes some useful library procedures and functions that you must declare EXTERN in order to use.

Wang PC Pascal implements three kinds of procedures and functions:

1. Some are predeclared, and the compiler translates them into other calls or special generated code (these you cannot pass as parameters).
2. Some are predeclared, but you call them normally (except for a name change).
3. Some are not predeclared but available as part of the Wang PC Pascal runtime library (these you must declare explicitly).

It is more useful when discussing these procedures and functions to categorize them by what they do rather than by how you implement them. Table 23-1 shows this categorization.

23.1 CATEGORIES OF AVAILABLE PROCEDURES AND FUNCTIONS

This section describes each of the categories shown in Table 23-1 and lists the procedures and functions each category includes. Each entry includes the syntax and a description, as well as examples and notes as appropriate.

Available Procedures and Functions

Table 23-1. Categories of Available Procedures and Functions

Category	Purpose
File system	Operate on files of different modes and structures
Dynamic allocation	Dynamically allocate and deallocate data structures on the heap at runtime
Data conversion	Convert data from one type to another
Arithmetic	Perform common transcendental and other numeric functions
Extend-level intrinsic procedures	Provide additional procedures and functions at the extend-level of Wang PC Pascal
System-level intrinsic procedures	Provide additional procedures and functions at the system level of Wang PC Pascal
String intrinsic procedures	Operate on STRING and LSTRING type data
Library	Available in the Wang PC Pascal runtime library: they are not predeclared. You must declare them with the EXTERN directive

23.1.1 File System Procedures and Functions

The Wang PC Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions fall into three categories, as shown in Table 23-2.

Available Procedures and Functions

Table 23-2. File System Procedures and Functions

Category	Procedures	Functions
Primitive	GET PAGE PUT RESET REWRITE	EOF EOLN
Text file I/O	READ READLN WRITE WRITELN	
Extend-level I/O	ASSIGN CLOSE DISCARD READSET READFN SEEK	

For details on each of these procedures and functions, see Chapter 24.

23.1.2 Dynamic Allocation Procedures

Two procedures, NEW and DISPOSE, allow dynamic allocation and deallocation of data structures at runtime. NEW allocates a variable in the heap, and DISPOSE releases it.

23.1.3 Data Conversion Procedures and Functions

Use the following procedures and functions to convert data from one type to another:

CHR	PACK	TRUNC
FLOAT	PRED	TRUNC4
FLOAT4	ROUND	UNPACK
ODD	ROUND4	WRD
ORD	SUCC	

Four of these convert any ordinal type to a particular ordinal type:

Available Procedures and Functions

- CHR (ordinal) to CHAR
- ODD (ordinal) to BOOLEAN
- ORD (ordinal) to INTEGER
- WRD (ordinal) to WORD

PRED and SUCC also operate on ordinal types.

Six of the conversion procedures and functions convert between INTEGER or INTEGER4 and REAL:

- FLOAT converts INTEGER to REAL
- FLOAT4 converts INTEGER4 to REAL
- ROUND converts REAL to INTEGER
- ROUND4 converts REAL to INTEGER4
- TRUNC converts REAL to INTEGER
- TRUNC4 converts REAL to INTEGER4

PACK and UNPACK transfer components between packed and unpacked arrays.

23.1.4 Arithmetic Functions

All arithmetic functions take a CONSTS parameter of type REAL4 or REAL8, or a type compatible with INTEGER (labeled "numeric" in the directory). ABS and SQR also take WORD and INTEGER4 values.

All functions on REAL data types check for an invalid (uninitialized) value. They also check for particular error conditions and generate a runtime error message if they find an error condition.

If the math-checking switch is on, errors in the use of the functions ABS and SQR on INTEGER, WORD, and INTEGER4 data generate a runtime error message. If the switch is off, the result of an error is undefined.

Table 23-3 lists the arithmetic function available, along with the routines the function calls, depending on whether it requires single- or double-precision.

Available Procedures and Functions

Table 23-3. Predeclared Arithmetic Functions

Name	Operation	REAL4	REAL8
ABS	Absolute value	(inline)	(inline)
ARCTAN	Arc tangent	ATSRQQ	ATDRQQ
COS	Cosine	CNSRQQ	CNDRQQ
EXP	Exponential	EXSRQQ	EXDRQQ
LN	Natural log	LNSRQQ	LNDRQQ
SIN	Sine	SNSRQQ	SNDRQQ
SQR	Square	(inline)	(inline)
SQRT	Square root	SRSRQQ	SRDRQQ

The Wang PC FORTRAN runtime library provides several additional REAL4 and REAL8 functions, as shown in Table 23-4. If you use them, you must declare them with the EXTERN directive.

Table 23-4. REAL Functions from the Wang PC FORTRAN Runtime Library

Operation	REAL4	REAL8
Arc cosine	ACSRQQ	ACDRQQ
Integral trunc	AISRQQ	AIDRQQ
Integral round	ANSRQQ	ANDRQQ
Arc sine	ASSRQQ	ASDRQQ
Arc tangent A/B	A2SRQQ	A2DRQQ
Hyperbolic cosine	CHSRQQ	CHDRQQ
Decimal log	LDSRQQ	LDDRQQ
Modulo	MDSRQQ	MDDRQQ
Minimum	MNSRQQ	MNDRQQ
Maximum	MXSRQQ	MXDRQQ
Power (REAL8**INTG4)		PIDRQQ
Power (REAL4**INTG4)		PISRQQ
Power (REAL ** REAL)	PRSRQQ	PRDRQQ
Hyperbolic sine	SHSRQQ	SHDRQQ
Hyperbolic tangent	THSRQQ	THDRQQ
Tangent	TNSRQQ	TNDRQQ

Some common mathematical functions are not standard in Pascal, but are relatively simple to accomplish with program statements or to define as functions in a program. Some typical definitions are:

```
SIGN (X)      is  ORD (X > 0) - ORD (X < 0)
POWER (X, Y)  is  EXP (Y * LN (X))
```

Available Procedures and Functions

You can also write your own functions in Wang PC Pascal to do the same thing. Defining functions like these is a good opportunity to use the PURE attribute (to obtain more efficient code). For example:

```
FUNCTION POWER (A, B: REAL): REAL [PURE];
BEGIN
  IF A <= 0 THEN
    ABORT ('Nonplus real to power', 24, 0);
    POWER := EXP (B * LN (A));
  END;
```

23.1.5 Extend-Level Intrinsic Procedures

The following intrinsic procedures and functions are available at the extend-level of Wang PC Pascal:

ABORT	EVAL	LOWORD
BYLONG	HIBYTE	RESULT
BYWORD	HIWORD	SIZEOF
DECODE	LOBYTE	UPPER
ENCODE	LOWER	

Several of these compose and decompose 1-byte, 2-byte, and 4-byte items: HIBYTE, LOBYTE, BYWORD, HIWORD, LOWORD, and BYLONG.

ENCODE and DECODE convert between internal and string forms of variables. ABORT invokes a runtime error.

The others, EVAL, LOWER, UPPER, RESULT, and SIZEOFF, are necessary in special situations (described for each function in Section 23.2).

23.1.6 System-Level Intrinsic Procedures

Several intrinsic procedures and functions are available at the system level:

FILLC	MOVESL
FILLSC	MOVESR
MOVE	RETYPE
MOVER	

The MOVE and FILL procedures perform low-level operations on byte strings. RETYPE changes the type of an expression arbitrarily.

23.1.7 String Intrinsic Procedures

The string intrinsic procedures feature provides a set of procedures and functions, some of which operate on STRINGS and LSTRINGS, and some on LSTRINGS only. Table 23-5 presents these procedures.

Available Procedures and Functions

Table 23-5. String Procedures and Functions

Name	Parameter
CONCAT	STRING
DELETE	STRING
INSERT	STRING
COPYLST	STRING
COPYSTR	STRING or LSTRING
POSITN	STRING or LSTRING
SCANEQ	STRING or LSTRING
SCANNE	STRING or LSTRING

23.1.8 Library Procedures and Functions

The following routines are not predeclared, but are available to you in the Wang PC Pascal runtime library. You must declare them, with the `EXTERN` directive, before you use them in a program.

- Initialization and termination routines -- You call `BEGOQQ` and `ENDOQQ` are called during initialization and termination, respectively. You can use them to invoke a debugger or to write custom messages, such as the time of execution, to the terminal screen. You can call `BEGXQQ` to restart a program and `ENDXQQ` to terminate it.
- Heap management routines -- Heap management routines complement the standard `NEW` and `DISPOSE` procedures and include:

```

ALLHQQ
FREECT
MARKAS
MEMAVL
RELEAS

```

- Interrupt routines -- These routines handle interrupt processing, although the actual effect varies with the target machine:

```

ENABIN
DISABIN
VECTIN

```

- Terminal I/O routines -- The following routines support direct input to and output from your terminal:

```

GTYUQQ
PTYUQQ
PLYUQQ

```

Available Procedures and Functions

- Semaphore routines -- The two procedures LOCKED and UNLOCK provide a binary semaphore capability. You can use them to ensure exclusive access of a resource in a concurrent system.
- No-overflow arithmetic functions -- These functions implement 16-bit and 32-bit modulo arithmetic. They return overflow or carry instead of invoking a runtime error.

LADDOK
LMULOK
SADDOK
SMULOK
UADDOK
UMULOK

- Clock routines -- These routines provide operating system clock information:

TIME
DATE
TICS

23.2 DIRECTORY OF FUNCTIONS AND PROCEDURES

This section contains a list of all available procedures and functions, both those that are predeclared and those library routines that you can use if you declare them EXTERN. Each entry includes the heading, the category to which the operation belongs, and a description of what the procedure or function does. Notes and examples are included as appropriate. The headings are the same for both REAL4 or REAL8, unless specifically stated otherwise.

Available Procedures and Functions

PROCEDURE ABORT (CONST STRING, WORD, WORD);

An extend-level intrinsic procedure that halts program execution in the same way as an internal runtime error. The STRING (or LSTRING) is an error message. The string parameter is a CONST, not a CONSTS parameter. The first WORD is an error code (see Appendix H for error code allocations); the second WORD can be anything. The second WORD sometimes returns a file error status code from the operating system.

The parameters, as well as any information about the machine state (program counter, frame pointer, stack pointer) and the source position of the ABORT call (if the \$LINE and/or \$ENTRY debugging switches are on), appear in a termination message or are available to the debugging package.

If the \$RUNTIME switch is on, error messages report the location of the procedure or function that called the routine that also called ABORT. If \$RUNTIME is on, \$LINE and \$ENTRY should be off, and routines in a source file should only call other \$RUNTIME routines.

FUNCTION ABS (X: NUMERIC): NUMERIC;

An arithmetic function that returns the absolute value of X. Both X and the return value are of the same numeric type: REAL4, REAL8, INTEGER, WORD, or INTEGER4. Since WORD values are unsigned, ABS (X) always returns X if X is of type WORD.

FUNCTION ACSROQ (CONSTS A: REAL4): REAL4; andFUNCTION ACDROQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the arc cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION AISROQ (CONSTS A: REAL4): REAL4; andFUNCTION AIDROQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the integral part of A, truncated toward zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION ALLHQQ (SIZE: WORD): WORD;

A library routine (heap management function) that returns zero if the heap is full, one if the heap structure is in error, or MAXWORD if the system interrupts the allocator. Otherwise, it returns the pointer value for an allocated variable with the size you requested.

Generally, you use ALLHQQ with the RETYPE function. For example:

```
P_VAR := RETYPE (P_TYPE, ALLHQQ (28));
{RETYPE converts the value returned by}
{ALLHQQ (28) to the type P_TYPE.}
{This value is assigned to P_VAR.}

IF WRD (P_VAR) < 2 THEN GO_ABORT;
{PVAR is then checked for a heap}
{full or heap structure error.}
```

FUNCTION ANSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION ANDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that, like AISRQQ and AIDRQQ, return the truncated integral part of A, but round away from zero. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION ARCTAN (X: REAL): REAL;

An arithmetic function that returns the arc tangent of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare ATSRQQ (CONSTS REAL4) and/or ATDRQQ (CONSTS REAL8) and use them instead.

FUNCTION ASSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION ASDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the arc sine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

PROCEDURE ASSIGN (VAR F; CONSTS N: STRING);

A file system procedure (extend-level I/O) that assigns an operating system file name in a STRING (or LSTRING) to a file F. See Section 24.3.1 for a description of ASSIGN.

FUNCTION ATSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION ATDRQQ (CONSTS A: REAL8): REAL8;

See FUNCTION ARCTAN

FUNCTION A2SRQQ (A, B: REAL4): REAL4; and
FUNCTION A2DRQQ (A, B: REAL8): REAL8;

Arithmetic functions that return the arc tangent of (A/B). Both A and B, as well as the return value, are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

PROCEDURE BEGOQQ:

A library routine (initialization) that you call during initialization. The default version does nothing. However, you can write your own version of BEGOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to a terminal screen. See also PROCEDURE ENDOQQ.

PROCEDURE BEGXQQ:

A library routine (initialization) that is the defined entry point for the load module after your program is linked and loaded.

As the overall initialization routine, BEGXQQ performs the following actions:

- It resets the stack and the heap.
- It initializes the file system.
- It calls BEGOQQ.
- It calls the program body.

BEGXQQ can be useful for restarting after a catastrophic error in a ROM-based system. However, invoking this procedure to restart a program does not take care of closing any open files. Similarly, it does not reinitialize variables a VALUE section originally set, or variables with the initialization switch on.

FUNCTION BYLONG (INTEGER-WORD, INTEGER-WORD): INTEGER4:

An extend-level intrinsic function that converts WORDS or INTEGERS (or the LOWORDs of INTEGER4s) to an INTEGER4 value. BYLONG concatenates its operands:

```
BYLONG (A, B) =
ORD (LOWORD (A)) * 65535 + WRD (HIWORD (B))
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

FUNCTION BYWORD (ONE-BYTE, ONE-BYTE): WORD:

An extend-level intrinsic function that converts bytes (or the LOBYTEs of INTEGERS or WORDs) to a WORD value. It takes two parameters of any ordinal type. BYWORD returns a WORD with the first byte in the most significant part and the second byte in the least significant part:

```
BYWORD (A, B) = LOBYTE(A) * 256 + LOBYTE(B)
```

If the first value is of type WORD, its most significant bit becomes the sign of the result.

FUNCTION CHR (X: ORDINAL): CHAR;

A data conversion function that converts any ordinal type to CHAR. The ASCII code for the result is ORD (X). This is an extension to the ISO standard, which requires X to be of type INTEGER. An error occurs if ORD (X) > 255 or ORD (X) < 0. However, the compiler catches the error only if the range-checking switch is on.

FUNCTION CHSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION CHDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the hyperbolic cosine of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

PROCEDURE CLOSE (VAR F);

A file system procedure (extend-level I/O) that performs an operating system close on a file, ensuring that the file access terminates correctly. See Section 24.3.1 for a description of CLOSE.

FUNCTION CNSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION CNDRQQ (CONSTS A: REAL4): REAL4;

See FUNCTION COS.

PROCEDURE CONCAT (VARS D: LSTRING; CONSTS S: STRING);

A string intrinsic procedure that concatenates S to the end of D. The length of D increases by the length of S. An error occurs if D is too small, that is, if UPPER (D) < D.LEN + UPPER (S).

PROCEDURE COPYLST (CONSTS S: STRING; VARS D: LSTRING);

A string intrinsic procedure that copies S to LSTRING D. The length of D is UPPER (S). An error occurs if the length of S is greater than the maximum length of D, that is, if UPPER (S) > UPPER (D).

PROCEDURE COPYSTR (CONSTS S: STRING; VARS D: STRING);

A string intrinsic procedure that copies S to STRING D. The remainder of D is blanks if UPPER (S) < UPPER (D). An error occurs if the length of S is greater than the maximum length of D, that is, if UPPER (S) > UPPER (D).

FUNCTION COS (X: NUMERIC): REAL;

An arithmetic function that returns the cosine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare CNSRQQ (CONSTS REAL4) and/or CNDRQQ (CONSTS REAL8) and use them instead.

Available Procedures and Functions

PROCEDURE DATE (VAR S: STRING);

A clock procedure that, if available, assigns the current date to its STRING (or LSTRING) variable. If you pass an LSTRING as the parameter, you must set the length you want before calling the procedure. The format depends on the target operating system.

FUNCTION DECODE (CONST LSTR: LSTRING, X:M:N): BOOLEAN;

An extend-level intrinsic function that converts the character string in the LSTRING to its internal representation and assigns this to X. If the character string is not a valid external ASCII representation of a value whose type is assignment-compatible with X, DECODE returns FALSE and the value of X is undefined.

DECODE works exactly the same as the READ procedure, including the use of M and N parameters (see Section 24.2.2 for a discussion of these parameters). When X is a subrange, DECODE returns FALSE if the value is out of range (regardless of the setting of the range-checking switch.) The compiler ignores leading and trailing spaces and tabs in the LSTRING. All other characters in the LSTRING must be part of the representation.

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION ENCODE.

PROCEDURE DELETE (VAR S: LSTRING; I, N: INTEGER);

A string intrinsic procedure that deletes N characters from S, starting with S[I]. An error occurs if you attempt to delete more characters starting at I than it is possible to delete, that is, if $S.LEN < (I + N - 1)$.

PROCEDURE DISBIN;

A library routine (interrupt) that, along with ENABIN and VECTIN, handles interrupt processing. DISBIN disables interrupts; ENABIN enables interrupts; VECTIN sets an interrupt vector. The effect of these procedures varies with the target machine. See Appendix A for information about your implementation.

PROCEDURE DISCARD (VAR F);

A file system procedure (extend-level I/O) that closes and deletes an open file. See Section 24.3.1 for a description of DISCARD.

PROCEDURE DISPOSE (VARS P: POINTER);

A dynamic allocation procedure (short form) that releases the memory the variable P points to uses. P must be a valid pointer; it cannot be NIL, uninitialized, or pointing at an already disposed heap item. The NIL check switch checks these.

P should not be a reference parameter or a WITH statement record pointer. However, this does not generate an error message. You can use a DISPOSE of a WITH statement record at the end of the WITH statement without problem.

If the variable is a super array type or a record with variants, you can use the short form of DISPOSE safely to release the variable, regardless of whether you allocate it with the long or short form of NEW. Using the short form of DISPOSE on a heap variable you allocated with the long form of NEW is an ISO-defined error Wang PC Pascal does not detect.

PROCEDURE DISPOSE (VARS P: POINTER; T1, T2, ... TN: TAGS);

The long form of DISPOSE (a dynamic allocation procedure) works the same as the short form. However, the long form checks the size of the variable against the size the tag field or array upper bound values T1, T2, ...TN imply. These tag values should be the same as defined in the corresponding NEW procedure. See also the SIZEOF function, which uses the same array upper bounds or tag value parameters to return the number of bytes in a variable.

PROCEDURE ENABIN;

A library routine (interrupt handling) that, along with DISBIN and VECTIN, handles interrupt processing. ENABIN enables interrupts; DISBIN disables interrupts; VECTIN sets an interrupt vector. The effect of these procedures may vary with the target machine. See Appendix A for information about your implementation.

FUNCTION ENCODE (VAR LSTR: LSTRING, X:M:N): BOOLEAN;

An extend-level intrinsic function that converts the expression X to its external ASCII representation and puts this character string into LSTR. ENCODE returns TRUE, unless the LSTRING is too small to hold the string generated. In this case, ENCODE returns FALSE and the value of the LSTR is undefined. ENCODE works exactly the same as the WRITE procedure, including the use of M and N parameters (see Section 24.2.4 for a discussion of these parameters).

X must be one of the types INTEGER, WORD, enumerated, one of their subranges, BOOLEAN, REAL4, REAL8, INTEGER4, or a pointer (address types need the .R or .S suffix).

In a segmented memory environment, the LSTR parameter must reside in the default data segment.

See also FUNCTION DECODE.

Available Procedures and Functions

PROCEDURE ENDOQQ;

A library procedure (termination) that you call during termination. The default version does nothing. However, you can write your own version of ENDOQQ, if you want, to invoke a debugger or to write customized messages, such as the time of execution, to a terminal screen. You call ENDOQQ after errors are processed, if ENDOQQ itself invokes an error, the result is an infinite termination loop. See also PROCEDURE BEGOQQ.

PROCEDURE ENDXQQ;

The termination procedure that is the overall termination routine and performs the following actions:

- It calls ENDOQQ.
- It terminates the file system (closing any open files).
- It returns to the target operating system (or whatever called BEGXQQ).

ENDXQQ may be useful for ending program execution from inside a procedure or function without calling ABORT. ENDXQQ corresponds to the HALT procedure in other versions of Pascal.

FUNCTION EOF: BOOLEAN; and FUNCTION EOF (VAR F): BOOLEAN;

A file system function that indicates whether the current position of the file is at the end of the file F for SEQUENTIAL and TERMINAL file modes. EOF with no parameters is the same as EOF (INPUT).

See Section 24.1.3 for a more complete description of EOF.

FUNCTION EOLN: BOOLEAN; and FUNCTION EOLN (VAR F): BOOLEAN;

A file system function that indicates whether the current position of the file is at the end of a line in the text file F. EOLN with no parameters is the same as EOLN (INPUT). See Section 24.1.3 for a description of EOLN.

PROCEDURE EVAL (EXPRESSION, EXPRESSION, ...);

An extend-level intrinsic procedure that evaluates expression parameters only, but accepts any number of parameters of any type. EVAL evaluates an expression as a statement; it commonly evaluates a function for its side effects only, without using the function return value.

FUNCTION EXSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION EXDRQQ (CONSTS A: REAL8): REAL8;

See FUNCTION EXP.

FUNCTION EXP (X: NUMERIC): REAL;

A arithmetic function that returns the exponential value of X (that is, e to the X). Both X and the return value are of type REAL. To force a particular precision, declare EXSRQQ (CONSTS REAL4) and/or EXDRQQ (CONSTS REAL8) and use them instead.

PROCEDURE FILLC (D: ADRMEM; N: WORD; C: CHAR);

A system-level intrinsic procedure that fills D with N copies of the CHAR C. This procedure does no bounds checking.

See also PROCEDURE FILLSC for segmented address types. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter. These are dangerous but sometimes useful procedures.

PROCEDURE FILLSC (D: ADSMEM; N: WORD; C: CHAR);

A system-level intrinsic procedure that fills D with N copies of the CHAR C. This procedure does no bounds checking.

See also PROCEDURE FILLC for relative address types. The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter. These are dangerous but sometimes useful procedures.

FUNCTION FLOAT (X: INTEGER): REAL;

A data conversion function that converts an INTEGER value to a REAL value. You do not need this function normally, since the compiler usually does INTEGER-to-REAL conversion automatically. Because the runtime package needs FLOAT, however, it is included at the standard level.

FUNCTION FLOAT4 (X: INTEGER4): REAL;

A data conversion function that converts an INTEGER4 value to a REAL value. The computer does this type conversion automatically; however, it is possible to lose precision. (Losing precision is not an error.)

FUNCTION FREECT (SIZE: WORD): WORD;

A library function that returns an estimate of the number of times NEW could allocate heap variables with length SIZE bytes. FREECT takes into account DISPOSE and adjacent free blocks; you generally use it with the SIZEOF function. However, it does not assume any stack space is needed. Because stack space generally is needed, reduce the value returned accordingly. For example:

```
IF FREECT (SIZEOF (REC, TRUE, 5)) > 2
  THEN DO_SOMETHING
```

Available Procedures and Functions

PROCEDURE GET (VAR F);

A file system procedure. GET either reads the currently pointed-to component of F to the buffer variable F and advances the file pointer, or sets the buffer variable status to empty. See Section 24.1.1 for a description of GET.

FUNCTION GTYUQQ (LEN: WORD; LOC: ADSMEM): WORD;

A library function (terminal I/O) that reads a maximum of LEN characters from the terminal keyboard and stores them in memory beginning at the address LOC. The return value is the number of characters GTYUQQ actually reads. GTYUQQ always reads the entire line you enter, but loses any characters typed beyond the end of the buffer length. For example:

```
LSTR.LEN := GTYUQQ (UPPER(LSTR), ADS LSTR(1));
```

Together with PTYUQQ and PLYUQQ, GTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the Wang PC Pascal file system. (See Section 9.2 for further information on Unit U.)

FUNCTION HIBYTE (INTEGER-WORD): BYTE;

An extend-level intrinsic function that returns the most significant byte of an INTEGER or WORD. Depending on the target processor, the most significant byte is the first or the second addressed byte of the word.

See also FUNCTION LOBYTE.

FUNCTION HIWORD (INTEGER4): WORD;

An extend-level intrinsic function that returns the high-order word of the four bytes of the INTEGER4. The sign bit of the INTEGER4 becomes the most significant bit of the WORD.

See also FUNCTION LOWORD.

PROCEDURE INSERT and (CONSTS S:STRING; VARS D:LSTRING; I:INTEGER);

A string intrinsic procedure that inserts S starting just before D [I]. An error occurs if D is too small, that is, if

```
UPPER (D) < UPPER (S) + D.LEN + 1
```

or if:

```
D.LEN < I
```

FUNCTION LADDOK (A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

A library routine (no-overflow arithmetic) that sets C equal to A + B. LADDOK is one of two functions that do 32-bit signed arithmetic without causing a runtime error, even if the arithmetic debugging switch is on (the other is LMULOK). Both LADDOK and LMULOK return TRUE if there is no overflow, and FALSE if there is overflow. These routines are useful for extended-precision arithmetic, or modulo 2³² arithmetic, or arithmetic based on user input data.

FUNCTION LDSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION LDDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the logarithm, base 10, of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION LMULOK (A, B: INTEGER4; VAR C: INTEGER4): BOOLEAN;

A library routine (no-overflow arithmetic) that sets C equal to A times B. LMULOK is one of two functions that perform 32-bit signed arithmetic without causing a runtime error on overflow (the other is LADDOK). Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both LMULOK and LADDOK return TRUE if there is no overflow, and FALSE if there is overflow. These routines are useful for extended-precision arithmetic, modulo 2³² arithmetic, or arithmetic based on user input data.

FUNCTION LN (X: REAL): REAL;

An arithmetic function that returns the logarithm, base e, of X. Both X and the return value are of type REAL. To force a particular precision, declare LNSRQQ (CONSTS REAL4) and/or LNDRQQ (CONSTS REAL8) and use them instead. An error occurs if X is less than or equal to zero.

FUNCTION LNSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION LNDRQQ (CONSTS A: REAL8): REAL8;

See FUNCTION LN.

FUNCTION LOBYTE (INTEGER-WORD): BYTE;

An extend-level intrinsic function that returns the least significant byte of an INTEGER or WORD. Depending on the target processor, the least significant byte is the first or the second addressed byte of the word.

See also FUNCTION HIBYTE.

FUNCTION LOCKED (VARS SEMAPHORE: WORD): BOOLEAN;

A library function (semaphore) that, if the semaphore was available, returns the value TRUE and sets the semaphore unavailable. Otherwise, if it was already locked, LOCKED returns FALSE. UNLOCK sets the semaphore available. As a binary semaphore, there are only two states.

See also PROCEDURE UNLOCK.

FUNCTION LOWER (EXPRESSION): VALUE;

An extend-level intrinsic function, LOWER takes a single parameter of one of the following types: array, set, enumerated, or subrange. The value LOWER returns is one of the following:

- The lower bound of an array
- The first allowable element of a set
- The first value of an enumerated type
- The lower bound of a subrange

LOWER uses the type, not the value, of the expression. The value LOWER returns is always a constant.

See also FUNCTION UPPER.

FUNCTION LOWORD (INTEGER4): WORD;

An extend-level intrinsic function that returns the low-order WORD of the four bytes of the INTEGER4.

See also FUNCTION HIWORD.

PROCEDURE MARKAS (VAR HEAPMARK: INTEGER4);

A library procedure (heap management) that parallels the MARK procedure in other versions of Pascal. MARKAS marks the upper and lower limits of the heap. The DISPOSE procedure is generally more powerful, but MARKAS may be useful for converting from other Pascal dialects.

In other versions of Pascal, the parameter is of a pointer type. However, Wang PC Pascal needs two words to save the heap limits, as the heap grows toward both higher and lower addresses in some implementations. Do not use the HEAPMARK variable as a normal INTEGER4 number; you should only set it by MARKAS and pass it to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, M for example, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To release heap space, simply invoke the procedure with RELEAS (M). MARKAS and RELEAS work as intended only if you never call DISPOSE.

FUNCTION MDSRQQ (CONSTS A, B: REAL4): REAL4; and
FUNCTION MDDRQQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions that are A modulo B, defined as:

$MDSRQQ(A, B) = A - AISRQQ(A/B) * B$
 $MDDRQQ(A, B) = A - AIDRQQ(A/B) * B$

Both A and B are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION MEMAVL: WORD;

A library function (heap management) that returns the number of bytes available between the stack and the heap. MEMAVL acts like the MEMAVAIL function in UCSD Pascal. If you previously used DISPOSE, MEMAVL may return a value less than the actual number of bytes available.

FUNCTION MMSRQQ (CONSTS A, B: REAL4): REAL4; and
FUNCTION MMDRQQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions that return the value of A or B, whichever is smaller. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

See also FUNCTION MXSRQQ and FUNCTION MXDRQQ.

PROCEDURE MOVEL (S, D: ADRMEM; N: WORD);

A system-level intrinsic procedure that moves N characters (bytes) starting at S† to D†, beginning with the lowest addressed byte of each array. Regardless of the value of the range- and index-checking switches, there is no bounds-checking. For example:

MOVEL (ADR 'New String Value', ADR V, 16)

See also PROCEDURE MOVESL for segmented address types. Use MOVEL and MOVESL to shift bytes left or when the address ranges do not overlap.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter.

PROCEDURE MOVER (S, D: ADRMEM; N: WORD);

A system-level intrinsic procedure that is like MOVE, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right. As with MOVE, there is no bounds-checking. For example:

```
MOVER (ADR V[0], ADR V[4], 12)
```

See also PROCEDURE MOVESR for segmented address types.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter.

PROCEDURE MOVESL (S, D: ADSMEM; N: WORD);

A system-level intrinsic procedure that moves N characters (bytes) starting at S to D, beginning with the lowest addressed byte of each array. Regardless of the value of the range- and index-checking switches, there is no bounds-checking. For example:

```
MOVESL (ADS 'New String Value', ADS V, 16)
```

See also PROCEDURE MOVER for relative address types. Use MOVE and MOVESL to shift bytes left or when the address ranges do not overlap.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter.

PROCEDURE MOVESR (S, D: ADSMEM; N: WORD);

A system-level intrinsic procedure that is like MOVESL, but starts at the highest addressed byte of each array. Use MOVER and MOVESR to shift bytes right. As with MOVESL, there is no bounds-checking. For example:

```
MOVER (ADR V[0], ADR V[4], 12)
```

See also PROCEDURE MOVER for relative address types.

The MOVE and FILL procedures take value parameters of type ADRMEM and ADSMEM, but because all ADR (or ADS) types are compatible, the ADR (or ADS) of any variable or constant can be the actual parameter.

FUNCTION MYSROQ (CONSTS A, B: REAL4): REAL4; and
FUNCTION MNDROQ (CONSTS A, B: REAL8): REAL8;

Arithmetic functions that return the value of A or B, whichever is larger. Both A and B are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

See also FUNCTION MNSROQ and MNDROQ.

PROCEDURE NEW (VARS P: POINTER);

A library procedure (heap management, short form) that allocates a new variable V on the heap and at the same time assigns a pointer to V to the pointer variable P (a VARS parameter). The pointer declaration of P determines the type of V. If V is a super array type, use the long form of the procedure instead. If V is a record type with variants, the compiler assumes variants giving the largest possible size and can assign any variant to P.

PROCEDURE NEW (VARS P: POINTER; T1, T2, ... TN: TAGS);

A library procedure (heap management, long form) that allocates a variable with the variant the tag field values T1 through Tn. The tag field values appear in the order in which they are declared. You can omit any trailing tag fields.

If all tag field values are constant, Wang PC Pascal allocates only the amount of space required on the heap, rounded up to a word boundary. Wang PC Pascal assumes the value of any omitted tag fields to be such that it can allocate the maximum possible size.

If some tag fields are not constant values, the compiler uses one of two strategies:

- It assumes that the first nonconstant tag field and all following tags have unknown values, and allocates the maximum size necessary.
- It generates a special runtime call to a function that calculates the record size from the variable tag values available. This depends on the implementation. A similar procedure applies to DISPOSE and SIZEOF.

You should set all tag fields to their proper values after the call to NEW and never change them. The compiler does not do any of the following:

- Assign tag values
- Check that they are initialized correctly
- Check that their value does not change during execution

According to the ISO standard, a variable created with the long form of `NEW` cannot be any of the following:

- Used as an expression operand
- Passed as a parameter
- Assigned a value

Wang PC Pascal does not catch these errors. Fields within the record can be used normally.

Assigning a larger record to a smaller one the compiler allocates with the long form of `NEW` wipes out part of the heap. This condition is difficult to detect at compile time. Therefore, in Wang PC Pascal, any assignment to a record in the heap that has variants uses the actual length of the record in the heap, rather than the maximum length.

An assignment to a field in an invalid variant, however, may destroy part of another heap variable or the heap structure itself. The compiler does not detect this error unless all tag values are explicit, the tag values are correct, and the tag-checking switch is on.

The extend-level allows pointers to super arrays. You use the long form of `NEW` as described above, except that you give array upper bound values instead of tag values. You must also give all upper bounds. Bounds can be constants or expressions; in any case, only the size required is allocated.

You cannot assign or compare any part of an array that such a pointer references except `LSTRING`s. You can pass the entire array as a reference parameter if the formal parameter is of the same super array type. You can use components of the array normally.

FUNCTION ODD (X: ORDINAL): INTEGER;

A data conversion function that tests the ordinal value `X` to see whether it is odd. `ODD` is `TRUE` only if `ORD (X)` is odd; otherwise it is `FALSE`.

FUNCTION ORD (X: VALUE): INTEGER;

A data conversion function that converts to `INTEGER` any value of one of the types shown in Table 23-6, according to the rules given.

Table 23-6. Conversion to INTEGER

Type of X	Return value
INTEGER	X
WORD <= MAXINT	X
WORD > MAXINT	X - 2 * (MAXINT + 1) (same 16 bits as at start)
CHAR	ASCII code for X
Enumerated	Position of X in the type definition, starting with 0
INTEGER4	Lower 16 bits (same as ORD(LOWORD(INTEGER4))
Pointer	Integer value of pointer

PROCEDURE PACK (CONSTS A: UNPACKED; I: INDEX; VARS Z: PACKED);

A data conversion procedure that moves elements of an unpacked array to a packed array. If A is an ARRAY [M..N] OF T and Z is a PACKED ARRAY [U..V] OF T, then PACK (A, I, Z) is the same as:

FOR J := U TO V DO Z [J] := A [J - U + I]

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; that is, the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range-checking switch controls checking of the bounds.

PROCEDURE PAGE; and PROCEDURE PAGE (VAR F);

A file system procedure that causes skipping to the top of a new page when the textfile F is printed. PAGE with no parameter is the same as PAGE (INPUT). See Section 24.1.4 for a description of PAGE.

FUNCTION PISROQ (CONSTS A: REAL4; CONSTS B: INTEGER4): REAL4; and
FUNCTION PIDROQ (CONSTS A: REAL8; CONSTS B: INTEGER4): REAL8;

Arithmetic functions that return a value that is $A^{**}B$ (A to the INTEGER power of B). A is of type REAL4 or REAL8, as shown. B is always of type INTEGER4. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

PROCEDURE PLYUQQ:

A library routine (terminal I/O) that writes an end-of-line character to the terminal screen. Together with GETYQQ and PTYUQQ, PLYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the Wang PC Pascal file system. See Section 9.2 for further information on Unit U.

FUNCTION POSITN

(CONSTS PAT:STRING; CONSTS S:STRING; I:INTEGER):
INTEGER;

A string intrinsic function that returns the integer position of the pattern PAT in S, starting the search at S [I]. If PAT is missing or if I > upper (S), the return value is 0. If PAT is the null string, the return value is 1. There are no error conditions.

FUNCTION PRED (X: ORDINAL): ORDINAL;

A data conversion function that determines the ordinal "predecessor" to X. The ORD of the result returned is equal to ORD (X) - 1. An error occurs if the predecessor is out of range or overflow occurs. The compiler detects these errors if appropriate debug switches are on.

FUNCTION PRSRQQ (A, B: REAL4): REAL4; and

FUNCTION PRDRQQ (A, B: REAL8): REAL8;

Arithmetic functions that return a value that is $A^{**}B$ (A to the REAL power of B). Both A and B are of type REAL4 or REAL8, as shown. An error occurs if $A < 0$ (even if B happens to have an integer value). These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

PROCEDURE PTYUQQ (LEN: WORD; LOC: ADSMEM);

A library routine (terminal I/O) that writes LEN characters, beginning at LOC in memory, to the terminal screen. For example:

```
PTYUQQ (8, ADS 'PROMPT: ');
```

Together with GETYQQ and PLYUQQ, PTYUQQ is useful for doing terminal I/O in a low-overhead environment. These functions are part of a collection of routines called Unit U, which implements the Wang PC Pascal file system. See Section 9.2 for further information on Unit U.

PROCEDURE PUT (VAR F);

A file system procedure that writes the value of the file buffer variable F^T to the currently pointed-to component of F and advances the file pointer. See Section 24.1.1 for a description of PUT.

PROCEDURE READ (VAR F; P1, P2, ... PN);

A file system procedure, READ reads data from files. The compiler defines both READ and READLN in terms of the more primitive operation, GET. See Section 24.2 for a description of READ.

PROCEDURE READFN (VAR F; P1, P2, ... PN);

A file system procedure (extend-level I/O), READFN is the same as READ (not READLN) with two exceptions:

1. File parameter F should be present.
2. If a parameter P is of type FILE, the compiler reads a sequence of characters forming a valid file name from F and assigns them to P in the same manner as ASSIGN.

The compiler reads parameters of other types in the same way as the READ procedure. See Section 24.3.1 for a description of READFN.

PROCEDURE READLN (VAR F; P1, P2, ... PN);

READLN is a text file I/O procedure. At the primitive GET level, without parameters, READLN (F) is equivalent to the following:

```
BEGIN
  WHILE NOT EOLN (F) DO GET (F);
  GET (F)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end of line. See Section 24.2 for a description of READ.

PROCEDURE READSET

(VAR F; VAR L: LSTRING; CONST S: SETOFCHAR);

A file system procedure (extend-level I/O), READSET reads characters and puts them into L, as long as the characters are in the set S and there is room in L. See Section 24.3.1 for a description of READSET.

PROCEDURE RELEAS (VAR HEAPMARK: INTEGER4);

A library routine (heap management) that parallels the RELEASE procedure in other versions of Pascal. RELEAS disposes of heap space past the area set with a previous MARKAS call. The DISPOSE procedure in Wang PC Pascal is generally more powerful, but RELEAS may be useful for converting from other Pascal dialects.

In other versions of Pascal, the parameter is of a pointer type. However, Wang PC Pascal needs two words to save the heap limits, as in some implementations the heap grows toward both higher and lower addresses. Do not use the HEAPMARK variable as a normal INTEGER4 number; it only set by MARKAS and pass it to RELEAS.

To use MARKAS and RELEAS, pass an INTEGER4 variable, M for example, as a VAR parameter to MARKAS. MARKAS places the bounds of the heap in M. To RELEAS heap space, simply invoke the procedure with RELEAS (M).

MARKAS and RELEAS work as intended only if you do not call DISPOSE.

PROCEDURE RESET (VAR F);

A file system procedure that resets the current file position to its beginning and does a GET (F). See Section 24.1.2 for a description of RESET.

FUNCTION RESULT (FUNCTION-IDENTIFIER): VALUE;

An extend-level intrinsic function that provides access to the current value of a function. RESULT can only appear within the body of the function itself or in a procedure or function nested within it.

FUNCTION RETYPE (TYPE-IDENT, EXPRESSION): TYPE-IDENT;

A system-level intrinsic function that provides a generic type of escape, and returns the value of the given expression as if it had the type named by the type identifier. The types the type identifier and the expression imply should usually have the same length, but this is not necessary. Component selectors (array index, fields, reference, etc.) can follow RETYPE for a structure. RETYPE is a dangerous type of escape and may not work as intended. For example:

```
TYPE COLOR = (RED, BLUE, GREEN);
S2      = STRING (2);
VAR C :#CHAR;
I, J :#INTEGER;
R :#REAL4;
TINT:#COLOR;
.
.
R := RETYPE (REAL4, 'abcd');
{Here, a 4-byte string literal is}
{converted into a real number.}
{REAL4 numbers also}
{require 4 bytes.}

TINT := RETYPE (COLOR, 2)
{Here, 2 is converted into a color.}
{which in this case is GREEN.}
{This is a relatively "safe" use}
{of the RETYPE function.}

C := RETYPE (S2, I) [J]
{Here, I is retyped into a two}
{character string. Then J selects}
{a single character of the string}
{which is assigned to C.}
```

There are two other ways to change type in Wang PC Pascal.

- First, you can declare a record with one variant of each type needed, assign an expression to one variant, and then get the value back from another variant. (This is an error the standard level does not detect. The relative mapping of variables is subject to change between different versions of the compiler.)
- Second, you can declare an address variable of the type you want and assign to it the address of any other variable (using ADR).

Each of these methods has its own quirks; you should avoid both whenever possible.

PROCEDURE REWRITE (F);

A file system procedure that resets the current file position to its beginning. See Section 24.1.2, "RESET and REWRITE," for a description of REWRITE.

FUNCTION ROUND (X: REAL): INTEGER;

An arithmetic function that rounds X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER. The effect of ROUND on a number with a fractional part of 0.5 varies with the implementation. Two examples are:

ROUND (1.6) is 2
ROUND (-1.6) is -2

An error occurs if $\text{ABS}(X + 0.5) \geq \text{MAXINT}$.

FUNCTION ROUND4 (X: REAL): INTEGER4;

An arithmetic function that rounds real X away from zero. X is of type REAL4 or REAL8; the return value is of type INTEGER4. The effect of ROUND4 on a number with a fractional part of 0.5 varies with the implementation. Two examples are:

ROUND4 (1.6) is 2
ROUND4 (-1.6) is -2

An error occurs if $\text{ABS}(X + 0.5) \geq \text{MAXINT4}$.

FUNCTION SADDOK

(A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

A library routine (no-overflow arithmetic) that sets C equal to A + B. SADDOK is one of two functions that perform 16-bit signed arithmetic without causing a runtime error on overflow (the other is SMULOK). Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Both SADDOK and SMULOK return TRUE if there is no overflow, and FALSE if there is overflow. These routines can be useful for extended-precision arithmetic, modulo 2¹⁶ arithmetic, or arithmetic based on user input data.

FUNCTION SCANEQ (LEN: INTEGER; PAT: CHAR; CONSTS S: STRING; I: INTEGER):
INTEGER;

A string intrinsic function that scans, starting at S [I], and returns the number of characters skipped. SCANEQ stops scanning when it finds a character equal to pattern PAT or when it skips LEN characters. If LEN < 0, SCANEQ scans backwards and returns a negative number. SCANEQ returns the LEN parameter if it finds no characters equal to pattern PAT or if I > UPPER (S). There are no error conditions.

FUNCTION SCANNE (LEN: INTEGER; PAT: CHAR; CONSTS S: STRING; I: INTEGER):
INTEGER;

A string intrinsic function that is like SCANEQ, but stops scanning when it finds a character not equal to pattern PAT. SCANNE scans, starting at S [I], and returns the number of characters skipped. SCANEQ stops scanning when it finds a character not equal to pattern PAT or when it skips LEN characters. If LEN < 0, SCANEQ scans backwards and returns a negative number. SCANEQ returns LEN parameter if it finds all characters equal to pattern PAT or if I > UPPER (S). There are no error conditions.

PROCEDURE SEEK (VAR F; N: INTEGER4);

A file system procedure (extend-level I/O). SEEK is used to randomly access components of direct files. In contrast to normal sequential files, DIRECT files are random access structures.

See Section 24.3 for details.

FUNCTION SHSRQQ (CONSTS A: REAL4): REAL4; and
FUNCTION SHDRQQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the hyperbolic sine of A. A is of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION SIN (X: NUMERIC): REAL;

An arithmetic function that returns the sine of X in radians. Both X and the return value are of type REAL. To force a particular precision, declare SNSRQQ (CONSTS REAL4) and/or SNDRQQ (CONSTS REAL8) and use them instead.

FUNCTION SIZEOF (VARIABLE): WORD; and
FUNCTION SIZEOF (VARIABLE, TAG1, TAG2, ... TAGN): WORD;

An extend-level intrinsic function that returns the size of a variable in bytes. This function sets tag values or array upper bounds as in the NEW and DISPOSE functions. If the variable is a record with variants and you use the first form, it returns the maximum size possible. If the variable is a super array, you must use the second form, which gives upper bounds.

FUNCTION SMULOK(A, B: INTEGER; VAR C: INTEGER): BOOLEAN;

A library routine (no-overflow arithmetic function) that sets C equal to A x B. SMULOK is one of two functions that perform 16-bit signed arithmetic without causing a runtime error on overflow (the other is SADDOK). Normal arithmetic may cause a runtime error, even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow, and FALSE if there is overflow. These routines can be useful for extended-precision arithmetic, modulo 2¹⁶ arithmetic, or arithmetic based on user input data.

FUNCTION SNSRQO (CONSTS A: REAL4): REAL4; and
FUNCTION SNDROO (CONSTS A: REAL8): REAL8;

See FUNCTION SIN.

FUNCTION SQO (X: NUMERIC): NUMERIC;

An arithmetic function that returns the square of X, where X is of type REAL, INTEGER, WORD, or INTEGER4.

FUNCTION SQRT (X): REAL

An arithmetic function that returns the square root of X, where X is of type REAL. To force a particular precision, declare SRSRQO (CONSTS REAL4) and/or SRDRQO (CONSTS REAL8) and use them instead. An error occurs if X is less than 0.

FUNCTION SRSRQO (CONSTS A: REAL4): REAL4; and
FUNCTION SRDRQO (CONSTS A: REAL8): REAL8;

See FUNCTION SQRT.

FUNCTION SUCC (X: ORDINAL): ORDINAL;

A data conversion function that determines the ordinal "successor" to X. The ORD of the returned result is equal to ORD (X) + 1. An error occurs if the successor is out of range or overflow occurs. The compiler detects these errors if appropriate debug switches are on.

FUNCTION THSRQO (CONSTS A: REAL4): REAL4; and
FUNCTION THDRQO (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the hyperbolic tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION TICS: WORD;

A library routine (clock function) that, if available, returns the value of an operating system timing location. The result is in a time interval, such as hundredths of a second, depending on the target operating system.

Available Procedures and Functions

PROCEDURE TIME (VAR S: STRING):

A library routine (clock function) that, if available, assigns the current time to its STRING (or LSTRING) variable. If the parameter is an LSTRING, you must set the length before you call the TIME procedure. The format depends on the target operating system.

See also PROCEDURE DATE.

FUNCTION TNSROQ (CONSTS A: REAL4): REAL4; and
FUNCTION TNDROQ (CONSTS A: REAL8): REAL8;

Arithmetic functions that return the tangent of A. Both A and the return value are of type REAL4 or REAL8, as shown. These functions are from the Wang PC FORTRAN runtime library and you must declare them EXTERN before use.

FUNCTION TRUNC (X: REAL): INTEGER;

An arithmetic function that truncates X toward zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER. Examples are:

TRUNC (1.6) is 1
 TRUNC (-1.6) is -1

An error occurs if $\text{ABS}(X - 1.0) \geq \text{MAXINT}$.

FUNCTION TRUNC4 (X: REAL): INTEGER4;

An arithmetic function that truncates real X towards zero. X is of type REAL4 or REAL8, and the return value is of type INTEGER4. Examples are:

TRUNC4 (1.6) is 1
 TRUNC4 (-1.6) is -1

An error occurs if $\text{ABS}(X - 1.0) \geq \text{MAXINT4}$.

FUNCTION UADDOK (A, B: WORD; VAR C: WORD): BOOLEAN;

A library routine (no-overflow arithmetic function) that sets C equal to $A + B$. UADDOK is one of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow (UMULOK is the other). Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. The following is the binary carry resulting from this addition of A and B:

WRD (NOT UADDOK (A, B, C))

Both UADDOK and UMULOK return TRUE if there is no overflow and FALSE if there is overflow. These routines are useful for extended-precision arithmetic, modulo 2^{16} arithmetic, or arithmetic based on user input data.

FUNCTION UMULOK (A, B: WORD; VAR C: WORD): BOOLEAN;

A library routine (no-overflow arithmetic function) that sets C equal to $A \times B$. UMULOK is one of two functions that do 16-bit unsigned arithmetic without causing a runtime error on overflow UADDOK is the other. Normal arithmetic may cause a runtime error even if the arithmetic debugging switch is off. Each routine returns TRUE if there is no overflow and FALSE if there is overflow. These routines are useful for extended-precision arithmetic, modulo 2^{16} arithmetic, or arithmetic based on user input data.

PROCEDURE UNLOCK (VAR SEMAPHORE: WORD);

A library routine (semaphore procedure) that sets the semaphore available. As a binary semaphore, only two states exist. You can call UNLOCK any number of times and can initialize the semaphore.

See also FUNCTION LOCKED.

PROCEDURE UNPACK

(CONSTS Z: PACKED; VARS A: UNPACKED; I: INDEX);

A data conversion procedure that moves elements from a packed array to an unpacked array. If A is an ARRAY [M..N] OF T, and Z is a PACKED ARRAY [U..V] OF T, the above call is the same as:

FOR J := U TO V DO A [J - U + I] := Z [J]

In both PACK and UNPACK, the parameter I is the initial index within A. The bounds of the arrays and the value of I must be reasonable; that is, the number of components in the unpacked array A from I to M must be at least as great as the number of components in the packed array Z. The range-checking switch controls checking of the bounds.

See also PROCEDURE PACK.

FUNCTION UPPER (EXPRESSION): VALUE;

An extend-level intrinsic function. UPPER, like LOWER, takes a single parameter of one of the following types: array, set, enumerated, or subrange. The value UPPER returns is one of the following:

- The upper bound of an array
- The last allowable element of a set
- The last value of an enumerated type
- The upper bound of a subrange

The value UPPER returns is always a constant, unless the expression is of a super array type. In this case, UPPER returns the actual upper bound of the super array. UPPER uses the type and not the value of the expression.

See also PROCEDURE LOWER.

PROCEDURE VECTIN (V: WORD; PROCEDURE I [INTERRUPT]):

A library routine (interrupt handling procedure) that is one of three procedures for processing interrupts. VECTIN sets an interrupt vector, so that it connects interrupts of type V to procedure I. (ENABIN enables interrupts and DISBIN disables interrupts.) The effect of these procedures and the meaning of V varies with the target machine. See Appendix A for information about your implementation.

FUNCTION WRD (X: VALUE): WORD;

A data conversion procedure that converts to WORD any of the types shown in Table 23-7, according to the rules given.

Table 23-7. Conversion to WORD

Type of X	Return Value
WORD	X
INTEGER \geq 0	X
INTEGER $<$ 0	X + MAXWORD + 1 (that is, same 16 bits as at start)
CHAR	ASCII code for X
Enumerated	Position of X in the type definition, starting with 0
INTEGER4	Lower 16 bits (that is, same as LOWORD(INTEGER4))
Pointer	Word value of pointer

PROCEDURE WRITE (VAR F; P1, P2, ... PN); and

PROCEDURE WRITELN (VAR F; P1, P2, ... PN);

File system-level intrinsic procedures that writes data to files. The compiler defines WRITE and WRITELN in terms of the more primitive operation PUT. WRITELN is the same as WRITE, except it also writes an end-of-line. See Section 24.2.3 for descriptions of these procedures.

CHAPTER 24 FILE-ORIENTED PROCEDURES AND FUNCTIONS

Chapter 22 introduced you to procedures and functions in general and described their use and construction. Chapter 23 described eight categories of procedures and functions that are available to you either because they are predeclared or because they are part of the Wang PC Pascal runtime library. Chapter 23 described all procedures and functions in detail, except those that relate to file input and output. This chapter discusses all of the file I/O procedures and functions, as well as lazy evaluation and concurrent I/O, two special Wang PC Pascal features that help you use files easily.

The Wang PC Pascal file system supports a variety of procedures and functions that operate on files of different modes and structures. These procedures and functions fall into the categories shown in Table 24-1.

Table 24-1. File System Procedures and Functions

Category	Procedures	Functions
Primitive	GET PAGE PUT RESET REWRITE	EOF EOLN
Text file I/O	READ READLN WRITE WRITELN	
Extend-level I/O	ASSIGN CLOSE DISCARD READSET READFN SEEK	

24.1 FILE SYSTEM PRIMITIVE PROCEDURES AND FUNCTIONS

This section describes the seven primitive file system procedures and functions that perform file I/O at the most basic level. Later descriptions of READ and WRITE procedures are defined in terms of the primitives GET and PUT. Two related topics are also discussed in this section: lazy evaluation and concurrent I/O. In the following descriptions, F is a file parameter (files are always reference parameters), and F↑ is the buffer variable.

In a segmented environment, all file variables these procedures operate on must reside in the default data segment. This restriction increases the efficiency of file system calls.

24.1.1 GET and PUT

The primitive procedures GET and PUT read to and write from the buffer variable, F↑. GET assigns the next component of a file to the buffer variable. PUT performs the inverse operation and writes the value of the buffer variable to the next component of the file F.

PROCEDURE GET (VAR F);

This procedure is a primitive file system-level intrinsic procedure. If there is a next component in the file F, then:

- The current file position advances to the next component.
- The procedure assigns the value of this component to the buffer variable F↑.
- EOF (F) becomes FALSE.

Advancing and assigning can be deferred internally, depending on the mode of the file. If no next component exists, then EOF (F) becomes TRUE and the value of F↑ becomes undefined. EOF (F) must be FALSE before GET (F), since reading past the end of file produces a runtime error. However, if F has mode DIRECT, EOF (F) can be TRUE or FALSE, since DIRECT mode permits repeated GET operations at the end of the file. If F↑ is a record with variants, the compiler reads the variant with the maximum size.

PROCEDURE PUT (VAR F);

A primitive file system-level intrinsic procedure that writes the value of the file buffer variable F↑ at the current file position and then advances the position to the next component. The following rules apply:

- For SEQUENTIAL and TERMINAL mode files, PUT is legal if the previous operation on F was a REWRITE, PUT, or other WRITE procedure, and if it was not a RESET, GET, or other READ procedure.

- For DIRECT mode files, PUT can occur immediately after a RESET or GET.

Exceptions to these rules cause errors to occur. The value of $F\uparrow$ is always undefined after a PUT.

In Wang PC Pascal, the value of $F\uparrow$ after a PUT (F) varies, depending on the type of file. EOF (F) must be TRUE before PUT (F), unless F is a DIRECT mode file. EOF (F) is always TRUE after PUT (F). If $F\uparrow$ is a record with variants, the compiler writes the variant with the maximum size.

24.1.2 RESET and REWRITE

The procedures RESET and REWRITE set the current position of a file to its beginning. RESET prepares for later GET and READ operations. REWRITE prepares for later PUT and WRITE operations.

PROCEDURE RESET (VAR F);

A primitive file system intrinsic procedure that resets the current file position to its beginning and does a GET (F). If the file is not empty, the procedure assigns the first component of F to the buffer variable $F\uparrow$, and EOF (F) becomes false. If the file is empty, the value of $F\uparrow$ is undefined and EOF (F) becomes true. RESET initializes a file F prior to reading it. For DIRECT files, writing can occur after RESET as well.

In Wang PC Pascal, a RESET closes the file and then opens it in a way that is dependent on the operating system. An error occurs if the file name is not set (as a program parameter or with ASSIGN or READFN) or if the operating system cannot find the file. If an error occurs during RESET, RESET closes the file, even if it opened the file correctly and the error came with the initial GET.

RESET (INPUT) occurs automatically when you initialize a program, but you can also do it explicitly. RESET on a file with mode DIRECT allows either reading or writing, but does not create the file automatically. In addition, the initial GET reads record number one on a DIRECT mode file.

An explicit GET (F) immediately following a RESET (F) assigns the second component of the file to the buffer variable. A READ (F, X) following a RESET (F) sets X to the first component of F, however, since READ (F, X) is "X := $F\uparrow$; GET (F)".

PROCEDURE REWRITE (VAR F);

A primitive file system-level intrinsic procedure that positions the current file to its beginning. The value of $F\uparrow$ is undefined and EOF (F) becomes TRUE. This is necessary to initialize a file F before writing (for DIRECT files, reading can occur after REWRITE too).

In Wang PC Pascal, a REWRITE closes the file and then opens it in a way that is dependent on the operating system. REWRITE creates the file if it does not exist in the operating system. If it does exist, its old value is lost (unless it has mode DIRECT). The file name must be set as a program parameter or with ASSIGN or READFN. If an error occurs during REWRITE, REWRITE closes the file. If possible, this does not affect an existing file with the same name, but some target operating systems delete the existing file.

REWRITE (OUTPUT) occurs automatically when you initialize a program, but you can also do it explicitly. REWRITE on a DIRECT mode file allows both reading and writing. REWRITE does not do an initial PUT the way RESET does an initial GET.

24.1.3 EOF and EOLN

The functions EOF and EOLN check for end-of-file and end-of-line conditions, respectively. They return a BOOLEAN result. In general, these values indicate when to stop reading a line or a file.

FUNCTION EOF: BOOLEAN; and FUNCTION EOF (VAR F): BOOLEAN;

A primitive file system intrinsic function that indicates whether the buffer variable F↑ is at the end of the file F for SEQUENTIAL and TERMINAL file modes. Therefore, if EOF (F) is TRUE, either the file is being written or the last GET has reached the end of the file.

With the DIRECT file mode, if EOF (F) is TRUE, either the last operation was a WRITE (the file may not be at the end in this case) or the last GET reached the end of the file.

EOF without a parameter is equivalent to EOF (INPUT). EOF (INPUT) is generally never TRUE, except in some operating systems where a particular terminal character generates an end-of-file status. Calling the EOF (F) function accesses the buffer variable F↑.

FUNCTION EOLN: BOOLEAN; and FUNCTION EOLN (VAR F): BOOLEAN;

A primitive file system intrinsic function that indicates whether the current position of the file is at the end of a line in the text file F after a GET (F). The file must have ASCII structure.

According to the ISO standard, calling EOLN (F) when EOF (F) is TRUE is an error. Wang PC Pascal detects this error in most cases. The file F must be a file of type TEXT.

If EOLN (F) is TRUE, the value of F↑ is a space, but the file is at a line marker. EOLN without a parameter is equivalent to EOLN (INPUT). Calling the EOLN (F) function accesses the buffer variable F↑.

24.1.4 PAGE

The PAGE procedure helps in formatting text files. It is not a necessary procedure in the way GET and PUT are.

PROCEDURE PAGE; and PROCEDURE PAGE (VAR F);

A primitive file system intrinsic procedure that causes skipping to the top of a new page when the text file F prints. Because PAGE writes to the file, the initial conditions described for PUT must be TRUE. The file must have ASCII structure. PAGE without a parameter is equivalent to PAGE (OUTPUT).

If F is not at the start of a line, PAGE (F) first writes a line marker to F. If F has mode SEQUENTIAL or DIRECT, then PAGE (F) writes a form feed, CHR (12).

24.1.5 Lazy Evaluation

Lazy evaluation in Wang PC Pascal makes it easier for the compiler to READ a file from a terminal. The ISO standard defines the procedure RESET with an initial GET. Although acceptable in Pascal's original batch processing, sequential file environment, this kind of read-ahead does not work for interactive I/O.

Lazy evaluation defers actual physical input (text files only) when the system evaluates a buffer variable. For example, when the compiler reads and resets a normal file, the RESET procedure calls the GET procedure, which sets the buffer variable to the first component of the file. If the file is a terminal, however, this first component does not yet exist. At a terminal, therefore, you must first type a character to accommodate the GET procedure. Only then does a prompt ask you for input.

Lazy evaluation eliminates this problem for text files by giving the file's buffer variable a special status value that is either "full" or "empty." The normal condition after a GET (F) is empty. The status is full after a buffer variable is assigned to or assigned from. Full implies that the buffer variable value is equal to the currently pointed-to component. Empty implies just the opposite, that the buffer variable value does not equal the value of the currently pointed to component and that the compiler has deferred input to the buffer variable. Table 24-2 summarizes these rules.

Table 24-2. Lazy Evaluation

Statement	Status	Action at call	Status on exit
GET (F)	Full	Point to next file component. Becomes EMPTY because value pointed to is not in buffer variable.	Empty
GET (F)	Empty	Load buffer variable with current file component, then point to next file component. Becomes EMPTY because value pointed to is not in buffer variable.	Empty
Reference to F↑	Full	Requires no action.	Full
Reference to F↑	Empty	Load buffer variable with current file component.	Full

RESET (F) first sets the status full and then calls GET, which sets the status to empty without any physical input. The following is an example of lazy evaluation with automatic REWRITE call:

```
{INPUT is automatically a text file.}
{RESET (INPUT); done automatically.}
WRITE (OUTPUT, "Enter number: ");
READLN (INPUT, FOO);
```

The automatic initial call to the RESET procedure calls a GET procedure, which changes the buffer variable status from full to empty. The first physical action to the terminal is the prompt output from the WRITE. READLN does a series of the following operations:

```
temp := INPUT↑;
GET (INPUT)
```

Physical input occurs when each INPUT↑ is fetched and the GET procedure sets the status back to empty.

READLN ends with the sequence:

```
WHILE NOT EOLN DO GET (INPUT);
GET (INPUT)
```

This operation skips trailing characters and the line marker. The EOLN function invokes the physical input. Entering the carriage return sets the EOLN status. Both the GET procedure in the WHILE loop and the trailing GET set the status back to empty. The last physical input in the sequence above is reading the carriage return.

24.1.6 Concurrent I/O

On operating systems that support it, concurrent I/O permits a GET or PUT procedure to initiate the I/O and immediately return to the calling program. You can only perform concurrent I/O on binary structure files.

The program can do computation while it is filling or emptying the buffer variable. The buffer variable has another special status value that can be "ready" or "busy." The program must wait until the status becomes ready before it can access the buffer variable. For example, the following program fragment reads the file IN_FILE, does some computation with the current value, and then writes it to the file OUT_FILE:

```

WHILE NOT EOF (IN_FILE) DO
  (Check for end of input and)
  {wait until IN_FILE↑ ready.}

BEGIN
  READ (IN_FILE, BUFF);
  (IN_FILE↑ is ready, so assign it to BUFF;)
  {start reading next component.}

  OPERATE (BUFF);
  {Go process value during READ and WRITE.}

  WRITE (OUT_FILE, BUFF);
  {Wait until OUT_FILE↑ is ready.}
  {then assign BUFF and start writing.}

END

```

The preceding example uses READ and WRITE procedures. The following two lines are equivalent:

```

READ (IN_FILE, BUFF)
BUFF := IN_FILE↑; GET (IN_FILE)

```

So are these two:

```

WRITE (OUT_FILE, BUFF)
OUT_FILE↑ := BUFF; PUT (OUT_FILE)

```

Concurrent I/O applies to the procedures GET and PUT as well as to the procedures READ and WRITE. In practice, it is unusual for the Wang PC Pascal runtime system to handle concurrency. See Appendix A for information regarding your implementation.

When accessing the buffer variable, either for lazy evaluation or concurrency, Wang PC Pascal generates an I/O system call. However, if the buffer variable is an actual reference parameter, the procedure or function using that parameter can do I/O to the same file, and these special calls cannot execute.

Passing any buffer variable as a reference parameter is an error in Wang PC Pascal, although only a warning message occurs. Calling GET or PUT has an undefined effect on a file buffer variable a reference parameter indirectly accesses. Assigning the address of a buffer variable to an address type variable is equally dangerous, as this bypasses the lazy evaluation and concurrency mechanisms.

24.2 TEXT FILE INPUT AND OUTPUT

Readable input and output in standard Pascal are done with text files. Text files are files of type TEXT and always have ASCII structure. Normally, the standard text files INPUT and OUTPUT appear as program parameters in the PROGRAM heading:

```
PROGRAM IN_AND_OUT (INPUT,OUTPUT);
```

Other text files usually represent some input or output device such as a terminal, a card reader, a line printer, or an operating system disk file. The extend-level allows you to use files other than program parameters.

To facilitate the handling of text files, you can use the four standard procedures READ, READLN, WRITE, and WRITELN as well as the procedures GET and PUT.

- READ and READLN -- The procedures READ and READLN read data from text files. READ and READLN are defined in terms of the more primitive operation, GET. The procedure READLN is very much like READ, except that it reads up to and including the end of line.
- WRITE and WRITELN -- The procedures WRITE and WRITELN write data to text files. WRITE and WRITELN are defined in terms of the more primitive operation, PUT. The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE.

These procedures are more flexible in the syntax for their parameter lists, allowing, among other things, for a variable number of parameters. Moreover, the parameters can be of types other than CHAR, in which case an implicit data conversion operation accompanies the data transfer. In some cases, parameters can include additional formatting values that affect data conversions.

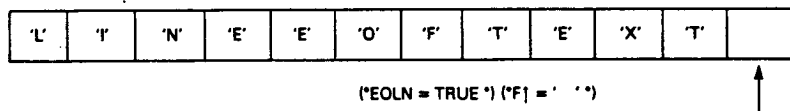
If the first variable is a file variable, it is the file the compiler reads from or writes to. Otherwise, the standard files INPUT and OUTPUT are the default values in the cases of reading and writing, respectively.

These two files have `TERMINAL` mode and `ASCII` structure and are predeclared as:

```
VAR INPUT, OUTPUT: TEXT;
```

Wang PC Pascal treats the files `INPUT` and `OUTPUT` like other text files. You can use them with `ASSIGN`, `CLOSE`, `RESET`, `REWRITE`, and the other procedures and functions. However, even if present as program parameters, a file name does not initialize them. Instead, the compiler assigns them to your terminal. `RESET` of `INPUT` and `REWRITE` of `OUTPUT` occur automatically, whether they are present as program parameters or not.

Text files represent a special case among file types insofar as they are structured into lines by "line markers". If, upon reading a text file `F`, the file position advances to a line marker (past the last character of a line), the value of the buffer variable `F↑` becomes a blank, and the standard function `EOLN (F)` yields the value `true`. For example:



Advancing the file position once more causes one of three things to happen:

- If the end of the file occurs, `EOF (F)` becomes `TRUE`.
- If the next line is empty, the compiler assigns a blank to `F↑` and `EOLN (F)` remains `TRUE`.
- Otherwise, the compiler assigns the first character of the next line to `F↑` and sets `EOLN (F)` to `FALSE`.

Since line markers are not elements of type `CHAR` in standard Pascal, only the procedure `WRITELN` can, in theory, generate them. In Wang PC Pascal, however, an actual character can be the line marker. It may therefore be possible to `WRITE` a line marker, but not to `READ` one.

When a text file that is being written to closes, the compiler appends a final line marker to the last line of any file that is not empty in which the last character is not already a line marker.

When a text file being read from reaches the end of a file that is not empty, a line marker for the last line is returned even if one was not present in the file. Therefore, lines in a text file always end with a line marker.

Any list of data a WRITELN writes can usually be read with the same list by a READLN (unless an LSTRING occurs that is not on the end of the list).

Interactive prompt and response is very easy in Wang PC Pascal. To have input on the same line as the response, use WRITE for the prompt. You must always use READLN for the response. For example:

```
WRITE ('Enter command: ');
READLN (response);
```

If you do not give a file, most of the text file procedures and functions assume either the INPUT file or the OUTPUT file. For example, if I is of type INTEGER, then READ (I) is the same as READ (INPUT, I).

24.2.1 READ and READLN

PROCEDURE READ and PROCEDURE READLN are file system intrinsic procedures for text file I/O. READ and READLN read data from text files. Both are defined in terms of the more primitive operation, GET. That is, if P is of type CHAR, then READ (F, P) is equivalent to:

```
BEGIN
  P := F↑;
  {Assign buffer variable F↑ to P.}
  GET (F)
  {Assign next component of file to F↑.}
END
```

READ can take more than a single parameter, as in READ (F, P1, P2, ... Pn). This is equivalent to the following:

```
BEGIN
  READ (F, P1);
  READ (F, P2);
  .
  .
  READ (F, Pn)
END
```

The procedure READLN is very much like READ, except that it reads up to and including the end-of-line. At the primitive GET level, without parameters, READLN is equivalent to the following:

```
BEGIN
  WHILE NOT EOLN (F) DO GET (F);
  GET (F)
END
```

A READLN with parameters, as in READLN (F, P1, P2, ... Pn), is equivalent to the following:

```
BEGIN
  READ (F, P1, P2, Pn);
  READLN (F)
END
```

READLN often skips to the beginning of the next line. You can use it only with text files (ASCII mode).

If you specify no other file, both READ and READLN read from the standard INPUT file. Therefore, you need not designate the name INPUT explicitly. For example, these two READ statements perform identical actions:

```
READ (P1, P2, P3)
{Reads INPUT by default}
READ (INPUT, P1, P2, P3)
```

At the standard level, parameters P1, P2, and P3 above must be of one of the following types:

```
CHAR
INTEGER
REAL
```

The extend-level also allows READ variables of the following types:

```
WORD
  an enumerated type
BOOLEAN
INTEGER4
  a pointer type
STRING
LSTRING
```

When the compiler reads a variable of a subrange type, the value must be in the range. If it is not, an error occurs, regardless of the setting of the range-checking switch.

The procedure READ can also read from a file that is not a text file (for example, has binary mode). You can use the form READ (F, P1, P2, ... Pn) on a binary file. This READ does not work as expected after a SEEK on a DIRECT mode file, however. For binary files, READ (F, X) is equivalent to:

```

BEGIN
  X := FT;
  GET (F)
END

```

24.2.2 READ Formats

The READ process for formatted types (everything except CHAR, STRING, and LSTRING) first reads characters into an internal LSTRING and then decodes the string to get the value. Formatted reads do the following:

- Skip leading spaces, tabs, form feeds, and line markers. For example, when doing READLN (I, J, K) where I, J, and K are integers, the numbers can all be on the same line or spread over several lines.
- Read characters as long as they are in the valid set of characters for the type. For example, the compiler reads "-1-2-3" as the string of characters for a single INTEGER, but an attempt to decode the string generates an error. This means that spaces, tabs, line markers, or illegal characters should separate items.
- Ignore M and N values in READ, except as noted for an N value with enumerated types. M and N parameters are illegal in binary reads.

Most of the formatting rules below apply to the function DECODE as well.

- INTEGER and WORD types -- If P is of type INTEGER, WORD, or a subrange of these, READ (F, P) implies reading a sequence of characters from F that form a number according to the normal Pascal syntax, and then assigning the number to P. READ (F, P) accepts nondecimal notation (16#C007, 8#74, 10#19, 2#101, #Face) for both INTEGER and WORD, with a radix of 2 through 36. If P is of an INTEGER type, READ (F, P) accepts a leading plus (+) or minus (-) sign. If P is of a WORD type, READ (F, P) accepts numbers up to MAXWORD (32768..65535).
- REAL and INTEGER4 types -- If P is of type REAL, or at the extend-level type INTEGER4, READ (F, P) implies reading a sequence of characters from F that form a number of the appropriate type and assigning the number to P. Nondecimal notation is illegal for REAL numbers, but is acceptable for INTEGER4 numbers. When reading a REAL value, a number with a leading or trailing decimal point is legal, even though this form gives a warning if it is a constant in a program.
- Enumerated and Boolean types -- At the extend-level, if P is an enumerated type or BOOLEAN, the compiler reads a number as a WORD subrange and assigns a value to P such that the number is the ORD of the enumerated type's value. In addition, if P is type BOOLEAN, reading one of the character sequences 'TRUE' or 'FALSE' assigns true and false, respectively, to P. The number must be in the range of the ORD values of the variable.

Also at the extend-level, if the parameter P is an enumerated type and includes the :N notation as in READ (P::N), characters are read from the file F that form a valid identifier or number. If the characters form a number, it is assumed to be the ORD value described in the previous paragraph, and if the characters form an identifier that is one of the enumerated type's constant identifiers, its value is assigned to P. Also, if the variable is BOOLEAN, reading one of the digits 1 or 0 assigns true or false to the BOOLEAN variable, and accepts 'TRUE' and 'FALSE' as the BOOLEAN constant identifiers.

Using the N notation directs the compiler to ignore the actual value of N, save the enumerated type's constant identifiers, and make them available to the applicable READ routine. Omitting the N notation saves memory that the identifiers would need.

- Reference types -- At the extend-level, if P is a pointer type, the compiler reads a number as a WORD and assigns to P in a way that depends on your implementation, so that writing a pointer and later reading it yields the same pointer value. The compiler reads the address types as WORDs using .R or .S notation.
- String types -- At the extend-level, if P is a STRING (n), the compiler reads the next "n" characters sequentially into P without skipping preceding line markers, spaces, tabs, or form feeds. If the compiler encounters the line marker before it reads n characters, it sets the remaining characters in P to blanks and the file position remains at the line marker.

If the STRING has n characters before the compiler encounters the line marker, the file position remains at the next character. A few implementations have a limit of 255 characters on the length of a STRING. P can be the super array type STRING (such as a reference parameter or pointer referent variable).

At the extend-level, if P is a LSTRING (n), the compiler reads the next "n" characters sequentially into P without skipping preceding line markers, spaces, tabs, or form feeds. If the compiler encounters the line marker before it reads n characters, it sets the remaining characters in P to blanks and the file position remains at the line marker.

If the STRING has n characters before the compiler encounters the line marker, the file position remains at the next character. A few implementations have a limit of 255 characters on the length of a STRING. P can be the super array type STRING (such as a reference parameter or pointer referent variable). READ (LSTRING) is useful when reading entire lines from a text file, especially when you need the length of the line. For example, the easiest way to copy a text file is by using READLN and WRITELN with an LSTRING variable.

Currently, READ and READLN do not use M field width parameters: you cannot read the line '123456' as two INTEGER numbers with READ (I:3, J:3). You can read two LSTRING (3) items, however, and then decode them to achieve the same effect.

24.2.3 WRITE and WRITELN

PROCEDURE WRITE and PROCEDURE WRITELN are file system-level intrinsic procedures (text file I/O) that write data to text files. WRITE and WRITELN are defined in terms of the more primitive operation, PUT; that is, if P is an expression of type CHAR and F is a file of type TEXT, then WRITE (F, P) is equivalent to:

```
BEGIN
  FT := P;
  {Assign P to buffer variable F↑}
  PUT (F)
  {Assign F↑ to next component of file}
END
```

WRITE can take more than one parameter, as in WRITE (F, P1, P2, ... Pn). This is equivalent to the following:

```
BEGIN
  WRITE (F, P1);
  WRITE (F, P2);
  .
  .
  WRITE (F, Pn)
END
```

The procedure WRITELN writes a line marker to the end of a line. In all other respects, WRITELN is analogous to WRITE. Thus, WRITELN (F, P1, P2, ... Pn) is equivalent to:

```
BEGIN
  WRITE (P1, P2, ... Pn);
  WRITELN (F)
END
```

If either WRITE or WRITELN has no file parameter, the default file parameter is OUTPUT. Therefore, the first statement in each of the following pairs is equivalent to the second:

```
WRITE (P1, P2, ... Pn)
WRITE (OUTPUT, P1, P2, ... Pn)

WRITELN (P1, P2, ... Pn)
WRITELN (OUTPUT, P1, P2, ... Pn)
```

At the standard level, parameters in a WRITE can be expressions of any of the following types:

```
CHAR      BOOLEAN
INTEGER   STRING
REAL
```

At the extend-level, expressions can also be of the following types:

```
WORD      (an enumerated type)
INTEGER4  (a pointer type)
LSTRING
```

Parameters may take optional M and N values (see Section 24.2.4 for information about M and N parameters).

Although the procedure WRITE can also write to a binary file (not a text file), do not use WRITE for DIRECT files after a SEEK operation because the complementary READ form does not work as you might expect.

For binary files, WRITE (F, X) is equivalent to:

```
BEGIN
  F := X;
  PUT (F)
END
```

The form WRITE (F, P1, P2, ... Pn) is also acceptable. Normally, binary writes do not accept M and N values.

24.2.4 WRITE Formats

In text files, data parameters to WRITE and WRITELN can take one of the following forms:

```
P      P:M    P:M:N    P::N
```

The M and N values can be considered value parameters of type INTEGER and format in various ways. The extend-level permits M and N values for both READs and WRITEs, and permits giving N without M, as in:

```
P::N
```

Using these levels in a nonstandard way is an error the standard level does not detect. Some cases use only M, or N, or neither; the compiler ignores unused M and N values.

Omitting M or N is the same as using the value MAXINT. For example, WRITE (12:MAXINT) uses the default M value (8 in this case). M and N values are not valid for binary files. In WRITE, the M value is the number of characters to write. In ISO Pascal, M must be greater than zero, and if the expression requires less than M characters, the compiler pads it on the left with spaces.

At the extend-level, M can also be negative or zero. If it is negative, the compiler uses the absolute value of M, but padding of spaces occurs on the right instead of the left. If it is zero, no characters are written. Wang PC Pascal does not detect these are ISO standard errors.

If the representation of the expression cannot fit in ABS (M) character positions, the compiler uses extra positions as needed for numeric types, or truncates the value on the right for string types. If M is not present or is equal to MAXINT, the compiler supplies a default value.

The N value signifies:

- The number of decimal places if P is of type REAL
- The output radix if P is of type INTEGER, WORD, INTEGER4, or pointer
- The numeric or identifier value if P is of an enumerated type

Most of the following formatting rules apply to the function ENCODE as well.

- INTEGER and WORD types -- If P is of type INTEGER, WORD, or a subrange of these, the compiler writes the decimal representation of P on the file, applying a leading minus sign if P is negative. WORD values are never negative. For INTEGER and WORD values, the default M value is 8.

If ABS (M) is smaller than the representation of the number, the compiler uses additional character positions as necessary. N writes in hexadecimal, decimal, octal, binary, or other base numbering using N equal to a number from 2 to 36; this is an extension to the ISO standard. If N is not 10 (or not present or MAXINT), then padding on the left is with zeros and not spaces. Omitting N or setting N to MAXINT or 10 implies a decimal radix.

WORD decimal numbers from 32768 to 65535 are written normally and not in their negative integer equivalents. Spaces or some other character not valid in numbers should separate values, so that the compiler reads values as separate numbers.

- REAL and INTEGER4 types -- If P is of type REAL, the compiler writes a decimal representation of the number P, rounded to the specified number of decimal places, to the file. If the N is missing or equal to MAXINT, the compiler writes a floating-point representation of P that consists of a coefficient and a scale factor to the file. If N is present, the compiler writes a rounded fixed point representation of P to the file, with N digits after the decimal point. If N is zero, P is a rounded integer, with a decimal point. The default value of M for REAL values is 14.

The following are examples of WRITE operations on REAL values:

This Statement	Produces This Output
WRITE (123.456)	' 1.2345600E+02'
WRITE (123.456:20)	' 1.2345600000000E+02'
WRITE (123.456::3)	' 123.456'
WRITE (123.456:2:3)	' 123.456'
WRITE (123.456:-20:3)	'123.456'

At the extend-level, if P is of type INTEGER4, the compiler writes the decimal representation of P on the file. The N value sets the radix, as in type INTEGER. The default M value is 14.

- Enumerated and Boolean types -- At the extend-level, if P is an enumerated type and N is not present or is equal to MAXINT, the compiler writes ORD (P) on the file as if it were a WORD. If N has the value 1, the compiler writes the enumerated type's constant identifier for the value of P on the file, as if it were a STRING. Using this N notation allocates memory for the enumerated type's constant identifiers.

At the standard level, if P is of type BOOLEAN, the compiler writes one of the strings 'TRUE' or 'FALSE' to the file as a STRING. The ORD value is never written for BOOLEAN types as it is for enumerated types (although you can use WRITE(ORD(P)) instead).

- Reference types -- At the extend-level, if P is a pointer type, the compiler writes P as a WORD. Writing a pointer and later reading it produces the same pointer value. The address types should appear as WORD values using .R or .S notation.
- String types -- If P is of type STRING (n), the compiler writes the value of P to the file. The default value of M is the length of the STRING n. If ABS (M) is less than the length of the string, the compiler writes only the first ABS (M) characters. If M is zero, it does not write anything. The compiler truncates the right portion of the STRING, even if M is negative.

At the extend-level, if P is of type LSTRING (n), the value of P is written to the file. The default value of M is the current length of the string, P.LEN. If ABS (M) is less than the current length, the compiler writes only the first ABS (M) characters. If M is zero, the compiler does not write anything. The compiler truncates the right portion of the LSTRING, even if M is negative. If ABS (M) is greater than the current length, spaces, not characters, fill the remaining positions past the length in the LSTRING. NULL:M can write a string of M blanks with NULL:M.

24.3 EXTEND-LEVEL I/O

At the extend-level, Wang PC Pascal has these additional I/O features:

- You can access three FCB fields: F.MODE, F.TRAP, and F.ERRS.
- A number of additional procedures are predeclared.
- Temporary files are available.

Section 16.6 discusses FCB fields in the context of files. The additional procedures and temporary files are described in the following sections.

File-Oriented Procedures and Functions

24.3.1 Extend-Level Procedures

The following are extend-level I/O procedures:

PROCEDURE ASSIGN (VAR F; CONSTS N: STRING)

A file system procedure (extend-level I/O) that assigns an operating system file name in a STRING (or LSTRING) to a file F. As a rule, ASSIGN truncates any trailing blanks. ASSIGN overrides any file name you set previously. You must set a file name before the first RESET or REWRITE on a file. ASSIGN on an open file (after RESET or REWRITE but before CLOSE) produces an error. ASSIGN to INPUT or OUTPUT is legal, but since these two files automatically open, you must close them before you assign to them.

PROCEDURE CLOSE (VAR F)

A file system procedure (extend-level I/O) that performs an operating system close on a file, ensuring that the file access terminates correctly. This is especially important for file variables allocated on the stack or the heap. Since these files must close before a RETURN or DISPOSE loses the file control block, they close automatically when a RETURN or DISPOSE releases stack or heap file variables.

File variables with the STATIC attribute in procedures and functions close automatically when the procedure or function returns. Files allocated statically at the program, module, or implementation level close automatically when the entire program terminates.

If necessary, when a CLOSE executes, a file the compiler is writing to has its operating system buffers flushed. However, the Wang PC Pascal buffer variable is not PUT. If the compiler is writing a file of type TEXT and the last line that is not empty does not end with a line marker, the compiler adds one to the end of the last line. If the file has the mode SEQUENTIAL and is being written, the compiler writes an end-of-file.

Some runtime errors may remove control from the Wang PC Pascal runtime system. In these cases, files being written cannot close, and the information in them is lost. A CLOSE on an already closed or an unopened file (no RESET or REWRITE) is legal. The compiler does not ignore CLOSE if error-trapping is on and there was a previous error. CLOSE turns off error-trapping for the file and clears the error status if the compiler detects no errors.

PROCEDURE DISCARD (VAR F);

A file system procedure (extend-level I/O) that closes and deletes an open file. DISCARD is much like CLOSE except DISCARD deletes the file.

PROCEDURE READFN (VAR F: P1, P2, ... PN);

A file system procedure (extend-level I/O); READFN is the same as READ (not READLN) with two exceptions:

File-Oriented Procedures and Functions

1. File parameter F should be present (INPUT is the default, but a warning occurs if you omit F).
2. If a parameter P is of type FILE, the compiler reads a sequence of characters that forms a valid file name from F and assigns them to P in the same manner as ASSIGN.

The compiler reads parameters of other types in the same way as the READ procedure.

READFN is like READ, not like READLN, and does not read the trailing line marker. If the first parameter in a READFN call is a file of any type, the compiler assumes it to be the text file. The file's name should not be read using INPUT as the default source.

READFN reads a program's parameters internally. It is useful when reading a file name and assigning the file name to a file in one operation.

PROCEDURE READSET (VAR F; VAR L: LSTRING, CONST S: SETOFCHAR);

A file system procedure (extend-level I/O), READSET reads characters and puts them into L, as long as the characters are in the set S and there is room in L. If no file parameter appears, this procedure assumes INPUT, as in READ and WRITE. This procedure always skips leading spaces, tabs, form feeds, and line markers.

Reading ceases at the first line marker, which is never in the type CHAR.

The runtime system uses READSET, along with ENCODE, to do the formatted READ procedures, as well as to read file names with READFN. READSET is useful when reading and parsing input lines for simple command scanners.

In a segmented memory environment, the L and S parameters must reside in the default data segment.

PROCEDURE SEEK (VAR F; N: INTEGER4);

A file system procedure (extend-level I/O), SEEK provides random access of DIRECT files. To use a DIRECT file, set the MODE field to DIRECT before a RESET or REWRITE opens the file; the file, F, must be a DIRECT mode file.

If the file is actually read or written sequentially, the usual READ and WRITE procedures are applicable.

SEEK modifies a field in file F so that the next GET or PUT applies to record number N. The record number parameter N can be of type INTEGER or WORD, as well as of type INTEGER4. For text files (ASCII structure), records are lines; for other files (binary structure), records are components. Record numbers start at one (not zero). If F is an ASCII file, SEEK sets the lazy evaluation status "empty." If F is a binary file, SEEK waits for I/O to finish and sets the concurrent I/O status "ready."

The following examples illustrate SEEK. Assume for instance, that a binary structured, DIRECT mode file contains the following CHAR contents:

	'A'	'B'	'C'	'D'	'E'	'F'	'G'	
N =	1	2	3	4	5	6	7	8

An implicit SEEK 1 occurs after a REWRITE or a RESET. Thus, with DIRECT mode files, the following sequences of commands might occur:

```
RESET (F);
{Initial SEEK 1, followed by GET;}
{F↑ now holds 'A'}
SEEK (F, 5);
{File position set to 5; F↑ still holds 'A'}
C := F↑
{C is now equal to 'A'; C does not equal 'E'}
```

The program does not assign the fifth component to C, as you might expect. To obtain this value, execute the following sequences of commands:

```
RESET (F);
{Initial SEEK 1, followed by GET;}
{F↑ now holds 'A'.}
SEEK (F, 5);
{File positioned at 5.}
GET (F);
{File buffer variable is loaded with 'E'.}
C := F↑
{C gets value 'E'.}
```

The rule to follow is to always follow a SEEK (F, N) with a GET to assure that the nth component is in the buffer variable.

GET and PUT operate normally on DIRECT mode files with binary structured files. READ and WRITE work only with ASCII files (text files). READ, in particular, does not work with DIRECT mode binary files because it assigns the buffer variable's value before it performs a GET. On the other hand, GET and PUT are not normally used with ASCII structured DIRECT mode files. Lazy evaluation makes READ and WRITE more appropriate. Take care when you mix normal sequential operations with DIRECT mode SEEK operations.

24.3.2 Temporary Files

Sometimes a program needs a scratch file for temporary, intermediate data. If this is the case, you can create a temporary file that is independent of the operating system. To do so without having to give the file a name in a specific format, assign a zero character as the name of the file. For example:

```
ASSIGN (F, CHR (0))
```

The file system creates a unique name for the file when it sees that you assigned the zero character as a name.

In environments where several running jobs are sharing a file directory, the job number is usually part of the name. RESET and REWRITE do not delete the file.

CHAPTER 25

COMPILABLE PARTS OF A PROGRAM

The Wang PC Pascal compiler can compile three kinds of source files: programs, modules, and implementations of units. You can, of course, compile modules and implementations of units separately and later link them to a program without recompilation. At the standard level, you can only compile entire programs; modules and units are Wang PC Pascal features available at the extend-level. The following is an example of a compilable program:

```
PROGRAM MAIN (INPUT, OUTPUT);
BEGIN
  WRITELN('Main Program')
END. {Main}
```

Example of a compilable module:

```
MODULE MOD_DEMO;
{No parameter list in heading}
PROCEDURE MOD_PROC;
BEGIN
  WRITELN
    ('Output from MOD_PROC in MOD_DEMO.')
END;
END. {Mod_Demo}
```

Example of a compilable unit:

```
INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  {UNIT_PROC is the only exported identifier}
  PROCEDURE UNIT_PROC;
END;
IMPLEMENTATION OF UNIT_DEMO;
  PROCEDURE UNIT_PROC;
  BEGIN
    WRITELN
      ('Output from UNIT_PROC in UNIT_DEMO.')
  END;
END. {Unit_Demo}
```

If you compile MODULE MOD_DEMO and UNIT UNIT_DEMO separately, you can later incorporate them into the main program as follows:

{INTERFACE required at the start of any}
 {source that implements or uses a unit.}

```
INTERFACE;
  UNIT UNIT_DEMO (UNIT_PROC);
  PROCEDURE UNIT_PROC;
END;

PROGRAM MAIN (INPUT, OUTPUT);
{USES clause below needed to connect}
{implementation and program.}
USES UNIT_DEMO;

{EXTERN declaration needed to connect}
{module's procedure.}
PROCEDURE MOD_PROC; EXTERN;
BEGIN
  Writeln('Output from Main Program.');
```

MOD_PROC;
 UNIT_PROC;
 END. {End of main program.}

After you compile the program MAIN, the output includes:

- Output from Main Program
- Output from MOD_PROC declared in MOD_DEMO
- Output from UNIT_PROC declared in UNIT_DEMO

The rules that govern the construction and use of programs, modules, and units are discussed in the following sections.

25.1 PROGRAMS

Except for its heading and the addition of a period at the end, a Pascal program has the same format as a procedure declaration. The statements between the keywords BEGIN and END are the body of the program. The following is an example of a program:

```
{Program heading}
PROGRAM ALPHA (INPUT, OUTPUT, A_FILE, PARAMETER);

{Declaration section}
VAR A_FILE: TEXT; PARAMETER: STRING (10);

{Program body}
BEGIN
  REWRITE (A_FILE);
  Writeln (A_FILE, PARAMETER);
END.
{Ends with period (.)}
```

Compilable Parts of a Program

The word "ALPHA" following the reserved word "PROGRAM" is the program identifier. The program identifier becomes the identifier for a PUBLIC procedure that lacks parameters, at a scope above all other identifiers in the program. This procedure also has the PUBLIC identifier ENTGQQ, which you call during initialization to start program execution.

You can call the program body as a PUBLIC procedure from another program, or from a module or unit, using the program identifier or ENTGQQ as the procedure name; this, however, is not good programming practice. This means that you can redeclare the program identifier within a program, and the usual scoping rules apply. The program identifier is at the same level as the predeclared identifiers, so giving a program an identifier such as INTEGER or READ generates an error message.

The program parameters denote variables set from outside the program. The program communicates with its environment through these variables.

At the standard level, all variables of any FILE type should be present as program parameters, since no other way to give an operating system file name to the file exists. At the extend-level, however, you can use the ASSIGN and READFN procedures to assign file names, so file variables need not appear as program parameters.

Program parameters differ entirely from procedure parameters; you do not pass these as parameters to the procedure that is the body of the program. You must declare all program parameters in the variable declaration part of the program block. If there are no program parameters or references to the files INPUT and OUTPUT, use the following form instead:

```
PROGRAM <identifier>;
```

The two standard files INPUT and OUTPUT receive special treatment as program parameters. Their values are not set as other program parameters. Do not declare these files, since they are already predeclared. Each should be present as a program parameter if you use them either explicitly or implicitly in the program:

```
WRITE (OUTPUT, 'Prompt: '); {Explicit use}
READLN (INPUT, P);

WRITE ('Prompt: ')          {Implicit use}
READLN (P);
```

The compiler gives a warning if you use them in the program but omit them as program parameters. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

You can redefine the identifiers INPUT and OUTPUT. All text file input and output procedures and functions (READ, EOLN, etc.) still use the original definition, however. RESET (INPUT) and REWRITE (OUTPUT) generate automatically, whether they are present as program parameters or not; you can also generate them explicitly.

Program initialization gives a value to every program parameter variable except INPUT and OUTPUT. Each parameter must be either of a simple type or of a STRING, LSTRING, or FILE type (any type the READFN procedure accepts). Program parameters must be entire variables; component selection is illegal.

Internally, each program parameter uses the file INPUT and generates READFN calls. Before each parameter is read, a special call is made to the internal routine PPMFQQ. PPMFQQ gets characters from an operating system interface routine called PPMUQQ, which gets them from the command line. PPMFQQ then puts those characters at the start of the file INPUT. The identifier of the parameter is passed to both routines (PPMFQQ and PPMUQQ). Some operating systems then use the identifier as a prompt.

The use of program parameters in Wang PC Pascal is best illustrated by showing how to change a program into a procedure. Begin with the following program:

```
PROGRAM ALPHA (INPUT, OUTPUT, P1, P2, ... Pn);
<declarations>
{Including those for P1, P2, ... Pn}
BEGIN
  <body>
END.
```

PROGRAM ALPHA can then become the following procedure:

```
PROCEDURE ENTGQQ [PUBLIC];
<declarations>
{Including those for P1, P2, ... Pn}
BEGIN
  PPMFQQ ('P1'); READFN (INPUT, P1);
  PPMFQQ ('P2'); READFN (INPUT, P2);
  .
  .
  PPMFQQ ('Pn'); READFN (INPUT, Pn);
  PFMEQQ ;
  {Called after all parameters are read}
  <program statements>
END;
```

The action of the interface routine PPMFQQ depends on the target operating system.

Some operating systems have elaborate mechanisms to manage this kind of parameter, using menus and default values. These mechanisms apply to Wang PC Pascal program parameters.

Other less sophisticated operating systems pass the remainder of the command line to a program that invoked it; in this case, the compiler reads parameter values from the command line.

If the operating system does not provide a program parameter mechanism, if an error occurs while you use such a mechanism, or if the operating system does not supply enough parameter values, then the PPMFQQ routine reverts to handling parameter values itself. It prompts you for every parameter with the parameter's identifier and reads the value you give it for the parameter. See Appendix A for details on how your implementation initializes program parameters.

25.2 MODULES

Modules provide a simple, straightforward method for combining several compilable segments into one program. Units (described in Section 25.3) provide a more powerful and structured method for achieving the same end.

Basically, a module is a program without a body. The identifier in the module heading has the same scope as a program identifier. The heading can also include attributes that apply to all procedures and functions in the module. There are no module parameters, nor is there a module body. A module ends with the reserved word END and a period. The following is an example of a module:

```
MODULE BETA [PUBLIC];      {Optional attributes}

PROCEDURE GAMMA;
  BEGIN WRITELN ('Gamma') END;

FUNCTION DELTA: WORD;
  BEGIN DELTA := 123 END;

END.                      {No body before END}
```

After the module identifier, you can give one or more attributes (in parentheses) to apply to all of the procedures and functions nested directly in the module. Depending on which, if any, attributes you specify, the following assumptions or restrictions apply:

- If no attribute list exists, the compiler assumes the PUBLIC attribute. However, if a list is present but empty, the compiler does not assume PUBLIC.
- The EXTERN directive with a particular procedure or function overrides the PUBLIC attribute for the entire module.
- You cannot give EXTERN and ORIGIN as attributes for an entire module, although you may specify them for individual procedures and functions.

- If you use PURE or INTERRUPT, the module must contain only functions for PURE and procedures for INTERRUPT.
- PUBLIC is the default attribute for all procedures and functions. In some cases, however, a PUBLIC procedure call has more overhead than a purely local one. In other cases, the identifier of a local procedure may conflict with a global identifier you pass to the Linker. To avoid these problems, use PUBLIC with individual procedures and functions, and use empty brackets for the entire module (for example, MODULE BETA []).

Although a module contains only declarations and no body, you can use it as a procedure that lacks parameters; that is, you can declare the module identifier as a procedure and call it from other programs, modules, or units. This module procedure (unlike a similar procedure for programs or units) is never called automatically, as there is no way for the compiler to know whether you have loaded a module and thus whether to generate a call to it.

In some cases, the compiler generates module initialization code that should execute by calling the module as an EXTERN procedure. If such code is necessary, the compiler gives the warning "Initialize Module". If you see this message, declare the module as a EXTERN procedure without parameters and call the procedure once before you access anything in the module. (You must do this if module declares any FILE variables.)

Given a module M that declares its own file variables, a program that uses M should look like this:

```
PROGRAM P (INPUT, OUTPUT)
.
PROCEDURE M; EXTERN;
BEGIN
  M;           {Runtime call initializes}
               {file variables.}
.
END.
```

If the module uses any interfaces that require initialization, the compiler generates a warning that you should declare the module EXTERN and call it as described above.

If module M does not contain any of its own file variables or use any initialized units, you need not invoke M as a procedure in the body of the program or declare it as an EXTERN procedure.

Variables within modules do not receive any attributes automatically. Except for the initialization of FILE variables mentioned above, variables within modules are the same as program variables.

25.3 UNITS

Wang PC Pascal units provide a structured way to access separately compiled modules. A unit consists of an interface and an implementation.

The interface appears at the front of an implementation of a unit and at the front of any program, module, interface, or implementation that uses a unit.

A unit contains constants, types, super types, variables, procedures, and functions, all of which you declare in the interface of the unit. Any program, module, or implementation or another interface can use an interface. An implementation contains the bodies of the procedures and functions in a unit, as well as optional initialization for the unit. The general scheme is shown in Figure 25-1.

INTERFACE; UNIT X; <identifier-declarations> END;	
<heading> USES X; <declarations> <optional-body> END.	IMPLEMENTATION OF X: <identifier- implementations> <optional-body> END.

Figure 25-1. A Wang PC Pascal Unit

When you use units, their interfaces go before everything else in a source file, either in an implementation or in the program, module, or other unit that uses it. In Figure 25-1, the interface is shared; the same interface exists in both the implementation source file and in the other source file. Conversely, any other program, module, or unit could USE UNIT X; similarly, there could be another IMPLEMENTATION OF X, in assembly language, for example.

By separating the interface from the implementation, you can write and compile a program before or while writing the implementation. You can also load a program with one of several implementations (for example, one in Wang PC Pascal or one in assembly language). You can often better organize a large Wang PC Pascal program as a main program and a number of units (parts of the Wang PC Pascal runtime system are organized in this way). However, only a program, module, interface, or implementation can use a unit, not an individual procedure or function.

A program, module, implementation, or interface that uses an interface must start with the source file for that interface. Generally, the interface source file is a separate file, and an \$INCLUDE metacommand at the start of the source file brings in the interface source itself during compilation. Because there is then only one master copy of the interface, this is easier and more reliable than physically inserting the interface everywhere you use it (and running the risk of ending up with several different versions).

In some applications, you may want several versions of the same interface. For example, a separate version of the Wang PC Pascal file control block interface exists for every target file system; the program copies the included file from the desired interface version before compilation. Naturally, every version must declare the common identifiers; each may also have some constant values for use in \$IF metacommands for the version-specific portions of the interface.

Suppose the file X.INT contains the INTERFACE for UNIT X in Figure 25-1. If that is so, the compilant using the unit and the implementation of the unit need only to include the interface file at the start of the source file, as shown in Figure 25-2.

{\$INCLUDE: 'X.INT'}	
<compilant-heading> USES X; <declarations> <optional-body> END.	IMPLEMENTATION OF X; <identifier- implementations> <optional-body> END.

Figure 25-2. Unit with File X.INT

A Wang PC Pascal source file of any kind contains zero or more unit interfaces, separated by semicolons, and followed by a program, a module, or an implementation, which is followed by a period. Each of these entities is a division. See Sections 25.3.1 and 25.3.2 for details about divisions.

A unit consists of the unit identifier, followed by a list of identifiers in parentheses. These identifiers are called the constituents of the unit and are the ones provided by a unit or required by a program, module, or other unit. The unit is preceded by the keyword UNIT for a provided unit or USES for a required one.

All unit identifiers in a source file must be unique. The identifiers in parentheses, however, may differ in the providing and requiring divisions. Correspondence between identifiers provided and required is by position in the list (similar to formal and actual parameters in procedures).

The identifier list in a USES clause is optional; if it is not present, the compiler uses the identifiers in the UNIT list by default. Giving different identifiers in a USES clause allows you to change the identifiers in case several different interfaces have identifier conflicts. You can combine multiple USES clauses; thus, the following statements are equivalent:

```
USES A; USES B; USES C;
USES A, B, C;
```

A unit can also introduce optional initialization code. The words BEGIN and END at the end of an interface imply such code. This code exists in an optional body in an implementation. The following is an example of a unit that introduces initialization code:

The program file, PLOTBOX:

```
{ $INCLUDE: 'GRAPHI' }
PROGRAM PLOTBOX (INPUT, OUTPUT);
  USES GRAPHICS (MOVE, PLOT);
  { MOVE and PLOT are USED identifiers. }
  BEGIN
    MOVE (0, 0);
    PLOT (10, 0); PLOT (10, 10);
    PLOT (0, 10); PLOT (0, 0);
  END.
```

The interface file, GRAPHI:

```
INTERFACE;
  UNIT GRAPHICS (BJUMP, WJUMP);
  { Exported identifiers are BJUMP and WJUMP. }
  { In the above PROGRAM, MOVE and PLOT }
  { are aliases for these identifiers. }
  PROCEDURE BJUMP (X, Y: INTEGER);
  PROCEDURE WJUMP (X, Y: INTEGER);
  { Procedure headings only above. }
  BEGIN
  { BEGIN implies initialization code. }
  END;
```

The implementation file:

```

{$INCLUDE:'GRAPHI'}
{$INCLUDE:'BASEPL'}
{The following implementation USES}
{the UNIT BASEPL. Thus, the interface}
{is included above and the unit}
{used below.}
IMPLEMENTATION OF GRAPHICS;
{Implementation is invisible to user.}
USES BASEPLOT;
  {Procedures BJUMP and WJUMP are}
  {implemented below.}
  {only the identifiers}
  {are given in the heading.}
  {The parameter lists are given}
  {in the interface.}
PROCEDURE BJUMP;
  BEGIN DRAWLINE (BLACK, X, Y) END;
PROCEDURE WJUMP;
  BEGIN DRAWLINE (WHITE, X, Y) END;
BEGIN
  {Begin initialization.}
  DRAWLINE (BLACK, 0, 0)
END.

```

The interface file, BASEPL:

```

INTERFACE;
UNIT BASEPLOT (BLACK, WHITE, DRAWLINE);
  {Other identifiers besides procedure}
  {identifiers can be exported.}
  {BLACK and WHITE are}
  {exported constant identifiers.}
TYPE RAINBOW = (BLACK, WHITE, RED, BLUE, GREEN);
PROCEDURE DRAWLINE (C: RAINBOW; H, V: INTEGER);
{No BEGIN; therefore, not an initialized unit.}
END;

```

A USES clause can occur only directly after a program, module, interface, or implementation heading. When the compiler encounters a USES clause, it enters each constituent identifier (from the UNIT clause or USES clause itself) in the symbol table. Identifiers for variables, procedures, and functions correspond with identifiers in the interface, which then become external references for the Linker.

If you compiled the sample program above, every reference to the procedure PLOT would generate an external reference to WJUMP. However, references to DRAWLINE would use the same identifier for the external reference.

You enter any constants and types (including any super array types) in the interface in the program's symbol table (along with the new identifier, if any). Thus, a type in an interface is identical to the corresponding type in the USES clause.

Record field identifiers are the same in the program, interface, and implementation. You must give enumerated type constant identifiers explicitly, if you need them; the enumerated type identifier does not imply them automatically. An interface cannot provide labels, since the target label of a GOTO must occur in the same division as the GOTO.

25.3.1 The Interface Division

The structure of an interface is as follows:

1. An interface section starts with the reserved word `INTERFACE`, an optional version number in parentheses, and a semicolon.
2. Next comes the keyword `UNIT`, the unit identifier, the list of exported (constituent) identifiers in parentheses, and another semicolon.
3. Any other units this interface requires come next, in `USES` clauses.
4. The last section is the actual declarations for all identifiers in the interface list, using the usual `CONST`, `TYPE`, and `VAR` sections and procedure and function headings, in any order. `LABEL` or `VALUE` sections cannot appear here.
5. The interface ends with `BEGIN END` if it has initialization, or just with `END` if it has no initialization.

Except for `ORIGIN`, which you cannot use in interfaces, you can give most available attributes to variables, procedures, and functions. Because the compiler gives the `PUBLIC` or `EXTERN` attribute or `EXTERN` directive automatically, you must not specify attributes that may conflict with it (such as `PUBLIC` and `EXTERN`).

Usually, the only identifiers you declare are the constituents, but other identifiers are legal. If the interface needs a call to initialize the unit, the keyword `BEGIN` generates the call. The interface ends with the reserved word `END` and a semicolon. The following is an example of an interface division:

```

INTERFACE (3);
  UNIT KEYFILE (FINDKEY, INSKEY, DELKEY, KEYREC);
  USES KEYPRIM (KFCB, KEYREC);

  PROCEDURE FINDKEY (CONST NAME: LSTRING;
    VAR KEY: KEYREC;
    VAR REC: LSTRING);
  PROCEDURE INSKEY (CONST REC: LSTRING;
    VAR KEY: KEYREC);
  PROCEDURE DELKEY (CONST KEY: KEYREC);
  PROCEDURE NEWKEY (CONST KEY: KEYREC);
BEGIN
  {Signifies initialized unit.}
END;

```

In this example, KEYREC is part of the unit KEYPRIM, but the program does not export it as part of the unit KEYFILE. KFCB is also part of the KEYPRIM unit, but the KEYFILE unit does not export it. NEWKEY is defined in the interface, but the KEYFILE unit does not export it. This is legal but pointless, since NEWKEY is unknown even in the implementation of the unit.

Memory available during compilation limits the number of identifiers the compiler can process. This limit can be a problem if you have many interfaces, especially interfaces that use other interfaces. The symptom of this problem is the error message "Compiler Out Of Memory". The message occurs before the final USES clause in the program, module, or implementation you are compiling. To cure this problem, reduce the number of identifiers in interfaces other interfaces use. For example, make a single interface that contains only types (and type-related constants) your other interfaces share, and only use this interface in the others.

If you include any file variables in the interface, you must initialize the unit. The compiler does not give the usual warning "Initialize Variable" when you declare a file in an interface. If your interface contains files, be sure to end it with BEGIN END to initialize these files.

25.3.2 The Implementation Division

You can compile an implementation of a unit separately from other programs, modules, or units, but you must compile it along with its interface. The structure of an implementation is as follows:

- An implementation of an interface starts with the reserved words IMPLEMENTATION OF, followed by the unit identifier and a semicolon.
- Next comes a USES clause for units it needs only for its own use.
- Then come the usual LABEL, CONSTANT, TYPE, VAR, and VALUE sections and all procedures and functions you mention as constituents (which must be in the outer block) or use internally, in any order.

VALUE and LABEL sections can appear in the implementation, but not in the interface. The following is an example of an implementation:

```

IMPLEMENTATION OF KEYFILE;
  USES KEYPRIM (KEYBLOCK, KEYREC);

  VAR KEYTEMP: KEYREC;

  PROCEDURE FINDKEY;
  BEGIN
    .
    {Code for FINDKEY}
    .
  END;

  PROCEDURE INKEY;
  BEGIN
    .
    {Code for INKEY}
    .
  END;

  PROCEDURE DELKEY;
  BEGIN
    .
    {Code for DELKEY}
    .
  END;

BEGIN
  .
  {Any initialization code goes here.}
  .
END.

```

You cannot declare in the implementation any constants, variables, and types you already declared in the interface. You can declare other private ones, however. Procedures and functions that are constituents of the unit do not include their parameter list (the interface implies it) or any attributes. (The interface implies the PUBLIC attribute unless you specify the EXTERN directive.)

You must define all procedures and functions in the interface in the implementation. However, you can give them the EXTERN directive so that several implementations (or an implementation and assembly code) can implement a single interface. All procedures and functions with the EXTERN directive must appear first; the compiler checks for this and issues an error message if the EXTERN directive is missing or misplaced.

You can implement a unit in assembly language, in which case all variables, procedures, and functions should generate public definitions for the loader. You can also implement units in other programming languages, such as Wang PC FORTRAN, or in a mixture of languages. If you do not implement the interface in Wang PC Pascal, you must give the interface the proper calling sequence attribute (and of course you must be familiar with calling sequences and internal representation of parameters).

Several Wang PC Pascal runtime units are a combination of Wang PC Pascal and assembly language. As mentioned, any implementation section that does not implement all interface procedures and functions must, at the start of the implementation, declare such procedures and functions to be EXTERN.

An implementation, like a program, may have a body. The body executes when you invoke the program that uses the unit, so any initialization the unit needs can occur. This includes internal initialization, such as file variable initialization, as well as user initialization code. If the source file contains several units, each implementation body is called in the order in which its USES clause appears in the source file. Initialization code for a unit executes only once, however, no matter how many clauses refer to it.

As in a program, the body is a list of statements enclosed with the reserved words BEGIN and END. At initialization time, the compiler checks whether the version number of the implementation's interface and the version number of the program's interface are the same. This checking prevents you from trying to run a program with obsolete implementations. If no version number appears, the compiler assumes it is zero.

The keyword BEGIN before the final END indicates a unit with initialization. If the word BEGIN is not present, the implementation must not have a body and no initialization takes place. Uninitialized units lack the following:

- User initialization code
- A guarantee of only one initialization
- A version number check

The format for an initialized implementation of a unit is similar to a program:

```
IMPLEMENTATION OF <unit-identifier>
<declarations>
BEGIN
    <body>    {Initialization code}
END.
```

The format for an uninitialized implementation of a unit is similar to a module:

```
IMPLEMENTATION OF <unit-identifier>
<declarations>
{No initialization code}
END.
```


If the implementation for an uninitialized unit declares any files or uses any interfaces that require initialization, the compiler warns you to initialize the implementation. Initialization occurs automatically if you add the keyword `BEGIN` to both the interface and the implementation. As with a module, you can declare an uninitialized unit to be a procedure with the `EXTERN` attribute and then initialize it by calling it from the program.

CHAPTER 26

WANG PC PASCAL METACOMMANDS

Metacommands make up the compiler control language. Metacommands are compiler directives that allow you to control such things as:

- Wang PC Pascal language level
- Debugging and error handling
- Optimization level
- Use of the source file during compilation
- Listing file format

You can specify one or more metacommands at the start of a comment; you should separate multiple metacommands with either spaces or commas. The compiler ignores spaces, tabs, and line markers between the elements of a metacommand. Thus, the following are equivalent:

```
{ $PAGE:12 }
{ $PAGE : 12 }
```

To disable metacommands within comments, place any character that is not a tab or space in front of the first dollar sign, as shown:

```
{x$PAGE:12}
```

You may change compiler directives during the course of a program. For example, most of a program might use \$LIST-, with a few sections using \$LIST+ as needed. Some metacommands, such as \$LINESIZE, normally apply to an entire compilation.

If you are writing Wang PC Pascal programs for use with other compilers, keep in mind the fact that metacommands are always nonstandard and rarely transportable.

Metacommands invoke or set the value of a metavariable. Metavariables are classified as typeless, integer, on/off switch, or string.

- You invoke typeless metavariables when you use them, as in \$EXTEND.
- You can set integer metavariables to a numeric value, as in \$PAGE:101.

Wang PC Pascal Metacommands

- You can set on/off switches to a numeric value so that a value greater than zero turns the switch on and a value equal or less than zero turns it off, as in \$MATHCK:1.
- You can set string metavariables to a character string value, such as with \$TITLE:'COM PROGRAM'.

Table 26-1 illustrates the notational conventions for the following metaccommand descriptions

Table 26-1. Metaccommand Notation

Notation	Meaning
	Metaccommand is typeless
+ or -	Metaccommand is an on/off switch + sets value to 1 (on) - sets value to 0 (off) + or - in heading indicates default
:<n>	Metaccommand is an integer
:'<text>'	Metaccommand is a string

String values in the metalanguage can be either a literal string or string constant identifier. Constant expressions are illegal for either numbers or strings, although you can achieve the same effect by declaring a constant identifier equal to the expression and using the identifier in the metaccommand.

In metaccommands only, Boolean and enumerated constants change to their ORD values. Thus, a Boolean false value becomes 0 and true becomes 1.

Appendix G contains a complete alphabetical listing of Wang PC Pascal metaccommands.

26.1 LANGUAGE LEVEL SETTING AND OPTIMIZATION

The metaccommands shown in Table 26-2 let you control the level (standard, extend, or system) at which the compiler processes your program and the degree of optimization. The Wang PC Pascal compiler may not implement all of the metaccommands this chapter describes; see Appendix A for details.

Table 26-2. Language and Optimization Level

Name	Description
\$EXTEND	Adds extend-level features
\$INTEGER:<n>	Sets the length of the INTEGER type
\$REAL:<n>	Sets the length of the REAL type
\$ROM	Gives a warning on static initialization
\$SIMPLE	Disables global optimizations
\$SIZE	Minimizes size of code generated
\$SPEED	Minimizes execution time of code
\$STANDARD	Enables standard level only
\$SYSTEM	Adds extend- and system-level features

The compiler issues a warning message if it encounters a feature whose level you have not enabled. The default setting is **\$EXTEND**, which permits structured extensions that are relatively safe and portable. It also requires you to explicitly request **\$SYSTEM** extensions, which are by their nature low-level, machine-dependent, and relatively unstructured.

\$INTEGER and **\$REAL** set the length (precision) of the standard **INTEGER** and **REAL** data types. You can only set **\$INTEGER** to 2 (the default), for 16-bit integers. You can, however, set **\$REAL** to either 4 or 8 (the default), to make type **REAL** identical to **REAL4** or **REAL8**, respectively.

The effect of the **\$SIZE** and **\$SPEED** metacommands varies with the version of the optimizer in your compiler. The default is **\$SIZE**. If you select **\$SIMPLE**, no optimization of any kind occurs. **\$SIZE**, **\$SPEED**, and **\$SIMPLE** are all mutually exclusive. If you set **\$ROM**, the compiler warns that it will not initialize static data in either of the following situations:

- At a **VALUE** section
- Everywhere static data initialization occurs due to **\$INITCK** (described in Section 26.2).

26.2 DEBUGGING AND ERROR HANDLING

The metacommads shown in Table 26-3 are for debugging and error handling. They also generate code to check for runtime errors.

Table 26-3. Debugging and Error Handling

Metacommad	Description
\$BRAVE+	Sends error messages and warnings to the screen
\$DEBUG-	Turns on or off all the debug checking (CK in metacommads below)
\$ENTRY-	Generates procedure entry/exit calls for debugger
\$ERRORS:<n>	Sets number of errors allowed per page (default is 25)
\$GOTO-	Flags GOTO statements as "considered harmful"
\$INDEXCK+	Checks for array index values in range, including super array indices
\$INITCK-	Checks for use of uninitialized values
\$LINE-	Generates line number calls for the debugger
\$MATHCK+	Checks for mathematical errors such as overflow and division by zero
\$NILCK+	Checks for bad pointer values
\$RANGECK+	Checks for subrange validity
\$RUNTIME-	Determines context of runtime errors
\$STACKCK+	Checks for stack overflow at procedure or function entry
\$TAGCK-	Checks tag fields in variant records
\$WARN+	Gives warning messages in listing file

If any check is on when the compiler processes a statement, the compiler performs tests relevant to the statement. A runtime error invokes a call to the runtime support routine, EMSEQQ (synonymous with ABORT). When you call EMSEQQ, the compiler passes the following information to it:

- An error message
- A standard error code
- An optional error status value, such as an operating system return code

EMSEQQ also has available:

- The program counter at the location of the error
- The stack pointer at the location of the error
- The frame pointer at the location of the error
- The current line number (if \$LINE is on)
- The current procedure or function name and the source file name in which you compiled the procedure or function (if \$ENTRY is on)

Each of these metacommands is discussed in more detail in the following paragraphs. You can also give most of the metacommands in this group as command line switches to the compiler. See Section 26.7 for details.

\$BRAVE+

Sends error messages and warnings to your terminal (in addition to writing them to the listing file). If the number of errors and warnings is more than can fit on the screen, the earlier ones scroll off and you must check the listing file to see them all.

\$DEBUG-

Turns on or off all of the debug switches (those that end with CK). You may find it useful to use \$DEBUG- at the beginning of a program to turn all checking off and then selectively turn on only the debug switches you want. Alternatively, you can use this metacommand to turn all debugging on at the start and then selectively turn off those you do not need as the program progresses. Some error checks are on and some off by default.

\$ENTRY-

Generates procedure and function entry and exit calls. This lets a debugger or error-handler determine the procedure or function in which an error occurred. Since this switch generates a substantial amount of extra code for each procedure and function, you should use it only when debugging. \$LINE+ requires \$ENTRY+; thus, \$LINE+ turns on \$ENTRY, and \$ENTRY- turns off \$LINE.

\$ERRORS:<n>

Sets an upper limit for the number of errors allowed per page. Compilation aborts if your program exceeds that number. The default is 25 errors and/or warnings per page.

\$GOTO-

Flags GOTO statements with a warning that they are "considered harmful." This warning is useful in either of the following circumstances:

- To encourage structured programming in an educational environment
- To flag all GOTO statements during the process of debugging

\$INDEXCK+

Checks that array index values, including super array indices, are in range. Since array indexing occurs so often, \$INDEXCK enables bounds checking separately from other subrange checking.

\$INITCK-

Checks for the occurrence of uninitialized values, such as the following:

- Uninitialized integers and 2-byte INTEGER subranges with the hexadecimal value 16#8000
- Uninitialized 1-byte INTEGER subranges with the hexadecimal value 16#80
- Uninitialized pointers with the value 1 (if \$NILCK is also on)
- Uninitialized REALs with a special value

The \$INITCK metaccommand generates code to perform the following actions:

- Set such values uninitialized when you allocate them
- Set the value of INTEGER range FOR-loop control variables uninitialized when the loop terminates normally

- Set the value of a function that returns one of these types uninitialized when you enter the function

\$INITCK never generates any initialization or checking for WORD or address types. Statically-allocated variables have their initial values when a program loads them. Also, \$INITCK does not check values in an array or record when a program uses the array or record itself.

\$INITCK assigns variables allocated on the stack or in the heap their initial values with generated code. \$INITCK does not initialize any of the following classes of variables:

- Variables mentioned in a VALUE section
- Variant fields in a record
- Components of a super array the NEW procedure allocates

\$LINE-

Generates a call to a debugger or error-handler for each source line of executable code. This allows the debugger to determine the number of the line in which an error occurred. Because this metaccommand generates a substantial amount of extra code for each line in a program, you should turn it on only when debugging. \$LINE+ requires \$ENTRY+, so \$LINE+ turns on \$ENTRY, and \$ENTRY- turns off \$LINE.

\$MATHCK+

Checks for mathematical errors, including INTEGER and WORD overflow and division by zero. \$MATHCK does not check for an INTEGER result of exactly -MAXINT-1 (#8000); \$INITCK does catch this value if a program assigns and later uses it.

Turning \$MATHCK off does not always disable overflow checking. There are, however, library routines that provide addition and multiplication functions that permit overflow (LADDOK, LMULOK, SADDOK, SMULOK, UADDOK, and UMULOK). See Section 23.2 for descriptions of each of these functions.

\$NILCK+

Checks for the following conditions:

- Dereferenced pointers whose values are NIL
- Uninitialized pointers if \$INITCK is also on
- Pointers that are out of range
- Pointers that point to a free block in the heap

\$NILCK occurs whenever you pass a pointer to the **DISPOSE** procedure. **\$NILCK** does not check operations on address types.

\$RANGECK+

Checks subrange validity in the following circumstances:

- Assignment to subrange variables
- **CASE** statements without an **OTHERWISE** clause
- Actual parameters for the **CHR**, **SUCC**, and **PRED** functions
- Indices in **PACK** and **UNPACK** procedures
- Set and **LSTRING** assignments and value parameters
- Super array upper bounds you pass to the **NEW** procedure

\$RUNTIME-

If the **\$RUNTIME** switch is on when you compile a procedure or function, the "location of an error" is the place where you called the procedure or function rather than the location in the procedure or function itself. This information normally appears on your terminal, but you could link in a custom version of **EMSEQQ**, the error message routine, to do something different (such as invoke the runtime debugger or reset a controller). For more information on error handling, see Chapter 9.

\$STACKCK+

Checks for stack overflow when entering a procedure or function and when pushing parameters larger than 4 bytes on the stack.

\$TAGCK-

Checks those tag values with identifiers when accessing a variant field.

\$WARN+

Sends warning messages to the listing file (this is the default). If this switch is off, only fatal errors appear in the source listing.

26.3 SOURCE FILE CONTROL

A small group of metacommands provides some measure of control over the use of the source file during compilation. These commands are listed in Table 26-4 and described in more detail below.

Table 26-4. Source File Control

Name	Description
<code>\$IF <constant></code> <code>\$THEN <text1></code> <code>\$ELSE <text2></code> <code>\$END</code>	Allows conditional compilation of <text1> source if <constant> is greater than zero
<code>\$INCLUDE: '<filename>'</code>	Switches compilation from current source file to source file named
<code>\$INCONST: <text></code>	Allows interactive setting of constant values at compile time
<code>\$MESSAGE: '<text>'</code>	Allows the display of a message the screen to indicate which version of a program is compiling
<code>\$POP</code>	Restores saved value of all metacommands
<code>\$PUSH</code>	Saves current value of all metacommands

Because the compiler keeps one look-ahead symbol, it actually processes metacommands that follow a symbol before it processes the symbol itself. This characteristic of the compiler can be a factor in cases such as the following:

```
CONST Q = 1;
{$IF Q $THEN}
{Q is undefined in the $IF.}
```

```
CONST Q = 1; DUMMY = 0;
{$IF Q $THEN}
{Now Q is defined.}
X := P↑;
{$NILCK+}
{NILCK applies to P↑ here.}
```

```

X := P↑;;
{NILCK doesn't apply to P.}
{$NILCK-}

```

\$IF <constant> \$THEN <text> \$END

Allows for conditional compilation of a source text. If the value of the constant is greater than zero, then source text following the \$IF processes; otherwise it does not. An \$IF \$THEN \$ELSE construction is also available, as in the following example:

```

{$IF MSDOS $THEN}
SECTOR = S12;
{$ELSE}
SECTOR = S128;
{$END}

```

To simulate an \$IFNOT construction, use the following form of the metaccommand:

```

$IF <constant> $ELSE <text> $END

```

The constant can be a literal number or constant identifier. The text between \$THEN, \$ELSE, and \$END is arbitrary; it can include line breaks, comments, other metacommands (including nested \$IFs), and so on. The compiler ignores any metacommands within skipped text, except, of course, corresponding \$ELSE or \$END metacommands. The following are examples using the metaconditional:

```

{$IF FPCHIP $THEN}
CODEGEN (FADDCALL,T1,LEFTP)
{$END}

{$IF COMPSYS $ELSE}
IF USERSYS THEN DOITTOIT
{$END}

```

\$INCLUDE

Allows the compiler to switch processing from the current source to the file named. When the compiler reaches the end of the included file, the compiler switches back to the original source and continues compilation. Resumption of compilation in the original source file begins with the line of source text that follows the line in which the \$INCLUDE occurred. Therefore, always put the \$INCLUDE metaccommand at the end of a line.

\$INCONST

Allows you to enter the values of the constants (such as those used in \$IFs) during compilation, rather than editing the source. This is useful when you use metaconditionals to compile a version of a source for a particular environment, customer, target processor, and so on. Compilation can be either interactive or batch-oriented. For example, the metaccommand \$INCONST:YEAR produces the following prompt for the constant YEAR:

```
Inconst: YEAR =
```

You need only give a response such as:

```
Inconst: YEAR = 1983
```

The compiler assumes the response to be of type WORD. The effect is to declare a constant identifier named YEAR with the value 1983. This interactive setting of the constant YEAR is equivalent to the constant declaration:

```
CONST YEAR = 1983;
```

You can also respond with a quoted string literal to create a constant of type STRING (n). For example, the source file metaccommand \$INCONST:HEADER prompts for a header. By enclosing a literal string constant in quotation marks, you declare a string constant:

```
Inconst: HEADER = 'Processor Version 2.75'
```

\$MESSAGE

Allows you to send messages to your terminal during compilation. This is particularly useful if you use metaconditionals extensively, for example, and need to know which version of a program is compiling. The following is an example of the \$MESSAGE metaccommand:

```
($MESSAGE:'Message on terminal screen!')
```

\$PUSH and \$POP

Allow you to create a metaenvironment you can store with \$PUSH and invoke with \$POP. \$PUSH and \$POP are useful in \$INCLUDE files for saving and restoring the metaccommands in the main source file.

26.4 LISTING FILE CONTROL

The metaccommands listed in Table 26-5 and described in this section allow you to format the listing file as you wish.

Table 26-5. Listing File Control Metacommands

Metacommand	Description
\$LINESIZE:<n>	Sets width of listing
\$LIST+	Turns on or off source listing. Errors are always listed
\$OCODE+	Turns on disassembled object code listing
\$PAGE+	Skips to next page. Does not reset line number
\$PAGE:<n>	Sets page number for next page (does not skip to next page)
\$PAGEIF:<n>	Skips to next page if less than n lines left on current page
\$PAGESIZE:<n>	Sets length of listing in lines. Default is 55
\$SKIP:<n>	Skips n lines or to end of page
\$SUBTITLE:'<text>'	Sets page subtitle
\$SYMTAB+	Sends symbol table to listing file
\$TITLE:'<text>'	Sets page title

\$LINESIZE:<n>

Sets the maximum length of lines in the listing file. See Appendix A for the default on your system.

\$LIST+

Turns on the source listing. Except for \$LIST-, metacommands themselves appear in the listing. Section 26.5 describes the format of the listing file.

\$OCODE+

Turns on the symbolic listing of the generated code to the object listing file. Although the format varies with the target code generator, it generally looks like an assembly listing, with code addresses and operation mnemonics. In many cases, the object listing file truncates the identifiers for procedure, function, and static variables.

\$PAGE+

Forces a new page in the source listing. The page number of the listing file increases by one automatically.

\$PAGE:<n>

Sets the page number of the next page of the source listing. \$PAGE:<n> does not force a new page in the listing file.

\$PAGEIF:<n>

Conditionally performs \$PAGE+, if the current line number of the source file plus n is less than or equal to the current page size.

\$PAGESIZE:<n>

Sets the maximum size of a page in the source listing. The default is 55 lines per page.

\$SKIP:<n>

Skips n lines or to the end of the page in the source listing.

\$SUBTITLE:'<subtitle>'

Sets the name of a subtitle that appears beneath the title at the top of each page of the source listing.

\$SYMTAB+

If on at the end of a procedure, function, or compiland, sends information about its variables to the listing file (for example, see lines 14 and 17 in the sample listing file in Section 26.5). The left columns contain the following:

- The offset to the variable from the frame pointer (for variables in procedures and functions)

- The offset to the variable in the fixed memory area (for main program and STATIC variables)
- The length of the variable

A leading plus or minus sign indicates a frame offset. This offset is to the lowest address the variable uses.

The first line of the \$SYMTAB listing contains the offset to the return address from the top of the frame (zero for the main program), and the length of the frame from the frame pointer to the end including front-end temporary variables. It does not include code generator temporary variables.

For functions, the second line contains the offset, length, and type of the value the functions return. The remaining lines list the variables, including their type and attribute keywords, as shown in Table 26-6.

Table 26-6 Symbol Table Notation

Keyword	Meaning
PUBLIC	Has the PUBLIC attribute
EXTERN	Has the EXTERN attribute
ORIGIN	Has the ORIGIN attribute
STATIC	Has the STATIC attribute
CONST	Has the READONLY attribute
VALUE	Occurs in a VALUE section
VALUEP	Is a value parameter
VARP	Is a VAR or CONST parameter
VARSP	Is a VARS or CONSTS parameter
PROCP	Is a procedural parameter
SEGMEN	Uses segmented addressing
REGIST	Parameter passed in register

\$TITLE: '<title>'

Sets the name of a title that appears at the top of each page of the source listing.

26.5 LISTING FILE FORMAT

The following discussion of listing file format is keyed to the sample listing in Figure 26-1.

Use	Title	PAGE	1
User	Subtitle	12/11/82	
		10:49:17	
JG	IC Line#	Source Line	MS-Pascal Version 3.0 10/82
00	1	PROGRAM foo; {syntab+}	
10	2	VAR i:integer; k:ARRAY [-9..0] OF integer;	
	2	Warning 156,Assumed ;↑	
20	3	FUNCTION bar (VAR j: integer): integer;	
20	4	VAR k: ARRAY [0..9] OF integer;	
20	5	BEGIN	
+ 21	6	GOTO 1; {jump forward}	
	6	Warning 281 Label Assumed Declared	
= 21	7	i := bar (j); {assign to global}	
	8	l: {label}	
/ 21	9	j := bar (i); {global to VAR parm}	
- 21	10	GOTO 1; {jump backward}	
* 21	11	RETURN; GOTO 1; {other jumps}	
% 21	12	i := bar (i); {other global reference}	
21	13	j:= bar (j); {no global references}	
10	14	END;	
	14	Warning 306 Function Assignment Not Found	
Syntab	14	Offset Length Variable - BAR	
	- 2	24 Return offset, Frame length	
	- 2	2 (function return):Integer	
	+ 4	2 J :Integer VarP	
	- 22	20 K :Array	
10	15	BEGIN	
11	16	i := bar (i);	
00	17	END.	
Syntab	17	Offset Length Variable	
	0	24 Return offset, Frame length	
	2	2 I :Integer	
	4	20 K :Array	
Errors	Warns	In Pass One	
1	2		

Figure 26-1. Listing File Format

Every page has a heading that includes such information as your title and subtitle, set with the metacommands \$TITLE and \$SUBTITLE, respectively. If these metacommands appear on the first source line, they take effect on the first page. The page number appears in the right side of the first line of the heading. In some versions, the date and time appear in the right side of the second and third line, respectively. You can set the page number with \$PAGE:<n> or start a new page with \$PAGE+.

The fourth line of the listing contains the column labels. The contents of the first three columns are:

- The JG column -- The JG column contains flag characters. Jump flags, which appear under the J, can contain one of the following characters:

- + forward jump (BREAK or GOTO a label in a higher line number)
- backward jump (CYCLE or GOTO a label in a lower line number)
- * other jumps (RETURN or a mixture of jumps)

Codes for global variables (not local to the current procedure or function) appear in the column under G:

- = assignment to a nonlocal variable
- / passing a nonlocal variable as a reference parameter
- % a combination of the two

- The IC column -- The IC column contains information about the current nesting levels. The digit under the I refers to the identifier (scope) level, which changes with procedure and function declarations, as well as with record declarations and WITH statements. The digit in the C column refers to the control statement level; this number changes with BEGIN and END pairs, as well as with CASE and END and REPEAT and UNTIL pairs. The number in this column is useful for finding missing END keywords.

If the compiler does not actively use a line, all these columns are blank. Thus you can locate a portion of the source the compiler accidentally skips due to an \$IF and \$END pair or comment.

- The Line column -- The Line column shows the line number of the line in the source file. An included file gets its own sequence of line numbers. If \$LINE is on, this line number and the source file name identify runtime errors.

Two kinds of compiler messages appear in the listing: errors and warnings. A compilation with any errors cannot generate code. A compilation with warnings only can generate code, but the result may not execute correctly. Warnings start with the word "Warning" and a number (see, for example, line 2 in the sample listing). Errors start with an error number (see line 14 in the sample listing). See Appendix H for a complete listing of all warning and error messages.

You can suppress warning messages with the metaccommand \$WARN-, but this is not good practice. The metaccommand \$BRAVE+ sends error and warning messages to your terminal (as well as to the listing file). If more messages exist than can fit on a single screen, the first ones scroll off.

An up arrow (↑) indicates the location of the error in the listing file. The message itself may appear to the left or right of the arrow after a dashed line.

The compiler sometimes does not detect an error until after the listing of the following line. In this case, the error message line number is not in sequence. Tabs are legal in the source and pass on to the listing without change. If the tab spacing is every eight columns, the error pointer (↑) is generally correct. However, an error pointer near the end of a line may be displaced if the following line has tabs.

If the compiler encounters an error from which it cannot recover, it gives the message "Compiler Cannot Continue!". This message appears if any of the following occurs:

- The keyword PROGRAM (or IMPLEMENTATION, INTERFACE, or MODULE) or the program, module, or unit identifier is missing.
- The compiler encounters an unexpected end-of-file.
- The compiler finds too many errors; the maximum number of errors per page exceeds the parameter in the \$ERRORS metaccommand (the default is 25).
- The identifier scope becomes too deeply nested. See Appendix A for the nesting level limit for your implementation.

When the compiler cannot continue, for whatever reason, it simply writes the rest of the program to the listing file with very little error checking.

26.6 COMMAND LINE SWITCHES

You can give many of the debugging and error-handling metaccommands in Section 26.1 as switches during compilation. You can give the switches either on the compiler command line or, in response to prompts, anywhere that spaces can go. Table 26-7 lists the metaccommands that are available as compiler switches. See Section 4.4 for more information on using switches.

Table 26-7. Command Line Switches

Switch	Metacommand	Description
/A	\$INDEXCK	Checks for array index values in range (including super array indices)
/D	\$DEBUG	Turns on all other switches, including \$ENTRY and \$LINE
/E	\$ENTR	Generates procedure entry and exit calls for the debugger
/L	\$LINE	Generates line number calls for error-checking
/I	\$INITCK	Checks for use of uninitialized values
/M	\$MATHCK	Checks for mathematical errors, such as overflow and division by zero
/N	\$NILCK	Checks for invalid pointer values, including NIL
/Q	\$DEBUG	Turns off all other switches, including \$ENTRY and \$LINE
/R	\$RANGECK	Checks for subrange validity, including assignments
/S	\$STACKCK	Checks for stack overflow at procedure or function entry
/T	\$TAGCK	Checks tag fields in variant records

The /Q switch, in combination with the others, provides an effective means of tailoring your compilation. First, turn off all of the other switches and then selectively turn on only the ones you want or need.

Appendices

APPENDIX A

VERSION SPECIFICS

This appendix describes the current implementation of the Wang PC Pascal language for the Wang Professional Computer. It discusses additions and restrictions to the language described in the Chapters 10 through 26, and identifies features of Wang PC Pascal that are not yet implemented.

A.1 IMPLEMENTATION ADDITIONS

The following additions have been made to the language described in Chapters 10 through 26:

1. You can declare the following function EXTERN:

```
FUNCTION DOSXQQ
  (COMMAND, PARAMETER: WORD): BYTE;
```

This function invokes the operating system, passing a command in the AH register and an additional parameter in the DX register. The BYTE function return value is identical to the value the operating system returns in AL, the accumulator.

The PUBLIC variables CRCXQQ and CRDXQQ contain the values of the CX and DX registers after the call. The value of CRCXQQ is also loaded into CX before the call.

Several operating system functions are particularly useful:

- a. DOSXQQ (1, 0);

Returns the next character you type. If you did not type a character, DOSXQQ waits for input. The compiler returns the ASCII value of the typed character, and the typed character appears on the screen.

b. DOSXQQ (2, WRD ('x'));

Outputs the character 'x' to your terminal. Ignore the function return value. If you enter the CONTROL + S and CONTROL + Q commands to stop and start scrolling, and the CONTROL + P command to toggle the printer, these functions execute. This function expands tabs.

c. DOSXQQ (6, 255);

Returns the next character you type on the keyboard, or zero, if you do not type a character. CONTROL + S, CONTROL + Q, and CONTROL + P are not treated specially. The terminal screen does not display the character typed.

d. DOSXQQ (6, WRD ('x'));

Outputs the character 'x' to your terminal. This is the same as DOSXQQ (2, WRD ('x')) above, except that CONTROL + S, CONTROL + Q, and CONTROL + P are not treated specially. Ignore the function return value in this case.

e. DOSXQQ (11, 0);

Returns screen status. Returns the value 255 if you typed a character; returns a 0 if you did not type a character. This function checks for a keypress condition without actually reading the character.

f. DOSXQQ (13, 0);

This function is not necessary on the Wang PC, but is provided for compatibility with other operating systems (CP/M and CP/M-86) where this function resets disk tables.

2. The following Wang PC Pascal file names are available to indicate devices:

Name	Description	DOS Code
USER	Console	1, 2, and 6
LINE	Auxiliary input	3, 4

Special file names, like CON and NUL, are also available (see The Wang Professional Computer Introductory Guide for details).

However, using CON for the terminal causes buffering of input and output data and precludes interactive input and output. Use the file name USER instead.

3. Program parameters are available. When a program starts, there is a prompt for every program parameter. You can also give program parameters on the command line with which you invoke the program. If a program requires more parameters than appear on the command line, the system will prompt you for those parameters.

For example, assume that you want to execute the following program:

```
PROGRAM DEMO (INFILE, OUTFILE, P1, P2, P3);
VAR INFILE, OUTFILE : TEXT;
    P1, P2, P3      : INTEGER;
BEGIN
.
.
END.
```

From the command line, you could run the program as follows:

```
A:DEMO DATA1.FIL DATA2.FIL 7 8 123
```

If you give only the first parameter on the command line, the compiler prompts you as follows (your responses are shown underlined):

```
A: DEMO DATA1.FIL
OUTFILE: DATA2.FIL 7
P2: 8
P3: 123
```

The compiler cannot read an LSTRING parameter value of NULL from the command line and assumes it to be missing. Enter it by pressing the RETURN key in response to the prompt.

4. The PUBLIC variable CESXQQ, which contains the segment register value for the start of the operating system data area, is available. This allows you to reference the command line, as follows:

```
VAR MSDATA: ADS OF LSTRING (80);
    CESXQQ [EXTERN] : WORD;
BEGIN
    MSDATA.S := CESXQQ; MSDATA.R := 128;
    {MSDATA now contains the command line.}
END;
```

This data area also contains, at offset 2, the upper memory limit, expressed as the segment (for example, paragraph) address of the first byte after available memory. The lower memory segment address is simply 4 KB paragraphs (for example, 64 KB) above the default data segment. For example:

```

VAR LOMADS, HIMADS, MSDATA: ADS OF WORD;
    CESXQQ [EXTERN] : WORD;
BEGIN
    LOMADS := ADS LOMADS;
    LOMADS.S := LOMADS.S + 4096;
    LOMADS.R := 0;
    {LOMADS is first available address.}

    MSDATA.S := CESXQQ; MSDATA.R := 2;
    HIMADS.S := MSDATA↑; HIMADS.R := 0;
    {HIMADS is first unavailable address.}
END;

```

5. Wang PC Pascal supports TIME, TICS, and DATE. TICS returns hundredths of seconds.
5. The object code listing is not an integral part of pass 2. If you want an object code listing, you must run the program PAS3.EXE. See Section 2.2 for details.
7. Four-byte functions now return values in DX:AX, not ES:BX. Also, real-value functions now use the long return mechanism (refer to Section 8.1).
8. Wang PC Pascal uses the IEEE real number format.
9. Bankers' rounding is used when truncating real numbers that end with .5; that is, odd numbers are rounded up to an even integer, even numbers are rounded down to an even integer. For example:

TRUNC(4.5) = 4

TRUNC (207.5) = 208

2 IMPLEMENTATION RESTRICTIONS

The following restrictions apply to this implementation of Wang PC Pascal:

1. Identifiers can have up to 31 characters. The compiler truncates longer identifiers.
2. Numeric constants can have up to 31 characters. Like identifiers, the compiler truncates numeric constants longer than 31 characters.

3. The Linker for this version of Wang PC Pascal truncates global identifiers to 31 characters.
4. The PORT attribute for variables is identical to the ORIGIN attribute. It does not use I/O port addresses.
5. The maximum level to which you can statically nest procedures is 15. Dynamic nesting of procedures is limited by the size of the stack.
6. The FORTRAN attribute does nothing. Wang PC Pascal and Wang PC FORTRAN share the same code generator and calling sequence. Wang PC FORTRAN parameters are always passed as Wang PC Pascal VARS parameters.
7. \$SIMPLE currently turns off common subexpression optimization. \$SIZE and \$SPEED turn it back on (and have no other effect).

A.3 UNIMPLEMENTED FEATURES

The following Wang PC Pascal features are not currently available, or are available only as discussed below.

1. OTHERWISE is illegal in RECORD declarations.
2. Code generates for PURE functions, but no checking occurs.
3. The extend-level operators SHL, SHR, and ISR are not available.
4. The ENABIN, DISBIN, and VECTIN library routines are not available. The compiler ignores the INTERRUPT attribute.
5. The compiler does not check for invalid GOTOs.
6. READ, READLN, and DECODE cannot have M and N parameters.
7. Enumerated I/O, permitting the reading and writing of enumerated constants as strings, is not available.
8. The metacommands \$TAGCK, \$STANDARD, \$EXTEND, and \$SYSTEM have no effect.
9. The \$INCONST metacommand does not accept string constants.

APPENDIX B

WANG PC PASCAL FEATURES AND THE ISO STANDARD

Wang PC Pascal generally conforms to the current ISO draft standard, but does not yet implement the proposed conformant array mechanism. This controversial method of passing arrays of different bounds as one parameter type has not been tested, and the details change from draft to draft. The conformant array scheme is not part of the ANSI/IEEE standard nor the ISO Level 0 standard.

The super array type in Wang PC Pascal provides conformant array parameters as well as dynamic length arrays allocated on the heap. Programs correct according to the ISO standard (Level 0) or to the ANSI/IEEE standard should run correctly, without changes, under Wang PC Pascal. Since Wang PC Pascal features introduce new reserved words and other elements, however, this goal is not entirely possible.

B.1 WANG PC PASCAL AND THE ISO STANDARD

The ISO standard defines a large number of error conditions, but allows a particular implementation to handle an error by documenting the fact that the error is "not caught." This and other differences between Wang PC Pascal and the ISO standard are described below. A Wang PC Pascal program that conforms or tests conformance to the ISO standard must have both the metacommands \$STANDARD and \$DEBUG on.

Wang PC Pascal allows the following minor extensions to the current ISO/ANSI/IEEE standard:

1. A question mark (?) as a substitute for the up arrow (↑)
2. The underscore (_) in identifiers

Because of the way the compiler binds identifiers, the new reserved words added at the extend and system levels cannot be identifiers at the standard level. A new directive, EXTERN, and new predeclared functions are standard in Wang PC Pascal.

The following list summarizes the current differences between Wang PC Pascal at the standard level and the current ISO/ANSI/IEEE standard:

1. The ISO standard requires a separator between numbers and identifiers or keywords.

In some cases, Wang PC Pascal does not require a separator between a number and an identifier or keyword (for example, "100mod" is acceptable as "100 mod").

2. The ISO standard does not allow passing a component of a PACKED structure as a reference parameter.

Wang PC Pascal specifically permits passing a CHAR element of a PACKED ARRAY [1..n] OF CHAR as a reference parameter. Passing a tag field as a reference is an error not caught. Passing other packed components gives the usual error.

3. The ISO standard does not include the text file line-marker character in the set of CHAR values.

Wang PC Pascal permits all 256 8-bit values as CHAR values; with some operating systems, a particular CHAR value (such as carriage return) is also the line marker character.

4. The ISO standard requires that a variant appear for all possible tag values.

Wang PC Pascal permits a variant record declaration in which you do not specify all tag values.

5. The ISO standard requires that an identifier have only one meaning in any scope.

In Wang PC Pascal, using an identifier and then redeclaring it in the same scope is an error not caught. For example:

```
CONST X=Y; VAR Y: CHAR;
```

This has two meanings for Y in the same scope. Wang PC Pascal generally uses the latest definition for an identifier. There is one ambiguous case: if you declare type FOO in one scope and in an inner scope TYPE P = ↑ FOO; FOO = type; FOO has two meanings and your intent is ambiguous. In this case, the compiler uses the later definition of FOO and issues a warning.

6. The ISO standard requires field width M to be greater than zero in WRITE and WRITELN procedures.

Wang PC Pascal treats $M < 0$ as if $M = \text{ABS}(M)$, but field expansion takes place from the right rather than the left. M can also be zero, to WRITE nothing. Text file READ(LN) and WRITE(LN) parameters can take both M and N parameters. The form "V::N" is legal. When writing an INTEGER, the N parameter sets the output radix; when reading or writing an enumerated type, the N parameter sets the ordinal number or constant identifier option.

7. The ISO standard does not allow a variable with the long form of NEW to be assigned, used in an expression, or passed as a parameter. However, this is difficult to check for during compilation and expensive to check at runtime.

Wang PC Pascal allows assignments to these variables using the actual length of the target variable. The ISO standard error is not caught.

8. The ISO standard does not allow use of the short form of DISPOSE on a structure allocated with the long form of NEW. The ISO standard only permits a variable allocated with the long form of NEW to be released with the long form of DISPOSE, and no tag fields should change between the calls.

Wang PC Pascal allows use of the short form of DISPOSE on a structure allocated with the long form of NEW, and does not check for changes in tag values.

9. The ISO standard declares that when a "change of variant" occurs (such as assigning a new tag value), all the variant fields become undefined.

Wang PC Pascal does not set the fields uninitialized after it assigns a new tag and does not catch use of a variant field with an undefined value.

10. The ISO standard does not allow a variable with an active reference (that is, the records of an executing WITH statement or an actual reference parameter) to be disposed (if a heap variable) or to change by a GET or PUT (if a file buffer variable).

Wang PC Pascal does not catch these as errors.

11. The ISO standard currently defines $I \text{ MOD } J$ as an error if $J < 0$ and the result of MOD is positive, even if I is negative.

Wang PC Pascal does not currently use the new draft standard semantics for the MOD operator. Do not use MOD in programs you intend to be portable unless both operands are positive.

- 12. The ISO standard at Level 1 defines conformant array.

Wang PC Pascal does not implement the conformant array concept in Level 1 of the ISO standard. Super arrays provide much the same functionality in a more flexible way.

- 13. The ISO standard requires the control variable of a FOR loop to be local to the immediate block. Any assignment to this control variable is an error.

Wang PC Pascal allows you to use nonlocal variables if the variable is STATIC, so either a local variable or one at the PROGRAM level can be a FOR statement control variable. Wang PC Pascal also does not detect an assignment to the control variable as an error if assignment occurs in a procedure or function FOR statement calls.

- 14. The ISO standard requires the CHR argument to be INTEGER.

Wang PC Pascal allows CHR to take any ordinal type.

B.2 SUMMARY OF WANG PC PASCAL FEATURES

This outline summarizes Wang PC Pascal additions to the ISO standard. Unless otherwise noted, all are at the extend-level.

B.2.1 Syntactic and Pragmatic Features

1. The metalanguage (standard level), as described in Chapter 26.

\$BRAVE	\$PAGEIF
\$DEBUG	\$PAGESIZE
\$ENTRY	\$POP
\$ERRORS	\$PUSH
\$EXTEND	\$RANGECK
\$GOTO	\$REAL
\$INCLUDE	\$ROM
\$INCONST	\$RUNTIME
\$INDEXCK	\$SIMPLE
\$INTICK	\$SIZE
\$IF \$THEN \$ELSE \$END	\$SKIP
\$INTEGER	\$SPEED
\$LINE	\$STACKCK
\$LINESIZE	\$STANDARD
\$LIST	\$SUBTITLE
\$MATHCK	\$SYMTAB
\$MESSAGE	\$SYSTEM
\$NILCK	\$TAGCK
\$OCODE	\$TITLE
\$OPTBUG	\$WARN
\$PAGE	

2. Extra listing (standard level)
 - a. Flags for jumps, globals, identifier level, control level, header, trailer
 - b. Textual error and warning messages
3. Syntactic additions
 - a. ! as comment to end of line
 - b. Square brackets equivalent to BEGIN/END
4. Nondecimal number notation
 - a. Numeric constants with # or nn# (where nn = 2..36)
 - b. DECODE/READ takes # notation
 - c. ENCODE/WRITE with N of 2, 8, 10, 16

5. Extended CASE range
 - a. For CASE statements and record variants
 - b. OTHERWISE for all other values
 - c. A..B for range of values

B.2.2 Data Types and Modes

1. WORD type, WRD function, MAXWORD constant
2. REAL4 and REAL8 types
3. INTEGER4 type, MAXINT4 const;
4. FLOAT4, ROUND4, and TRUNC4 functions
5. Address types (system-level)
 - a. ADR and ADS types and operators
 - b. VARS and CONSTS parameters
6. SUPER array types
 - a. Conformant parameters
 - b. Dynamic length heap variables
 - c. Multidimensional super arrays
 - d. STRING and LSTRING super types
7. LSTRING type, NULL constant, .LEN field
8. Explicit byte offsets in records (system level)
9. CONST and CONSTS reference parameters for constants and expressions
10. Structured (array, record, and set) constants
11. Extended functions returning any assignable type
12. Variable selection on values returned from functions

13. Attributes:

EXTERN	PORT
EXTERNAL	PUBLIC
FORTRAN	PURE
INTERRUPT	READONLY
ORIGIN	STATIC

B.2.3 Operators and Intrinsic procedures

1. Extend-level operators

- a. Shift operators: SHL SHR ISR
- b. Bitwise logical: AND OR NOT XOR
- c. Set operators: < >

2. Constant expressions

- a. String constant concatenation with * operator
- b. Numeric, ordinal, Boolean expressions in type clauses
- c. Other constant functions:

CHR	UPPER
DIV	WRD
HIBYTE	*
HIWORD	+
LOBYTE	-
LOWER	<
LOWORD	<=
MOD	<>
ORD	=
RETYPE	>
SIZEOF	>=

3. Additional intrinsic functions at extend level

ABORT	LOWORD
BYLONG	RESULT
BYWORD	SIZEOF
DECODE	UPPER
ENCODE	HIWORD
EVAL	LOBYTE
HIBYTE	LOWER

4. Additional intrinsic functions at system level

FILLC	MOVESL
FILLSC	MOVESR
MOVEL	RETYPE
MOVER	

5. Intrinsic functions that operate on strings

- a. For STRING or LSTRING: COPYSTR, POSITN, SCANEQ, and SCANNE
- b. For LSTRING only: CONCAT, INSERT, DELETE, and COPYLST

6. Wang PC FORTRAN REAL library functions (standard level)

7. Wang PC Pascal library functions (standard level)

ALLHQQ	MARKAS
BEGQQQ	MEMAVL
BEGKQQ	PLYUQQ
DATE	PTYUQQ
DISBIN	RELEAS
ENDOQQ	SADDOK
ENDXQQ	SMULOK
ENABIN	TICS
FREET	TIME
GTUQQ	UADDOK
LADDOK	UMULOK
LMULOK	UNLOCK
LOCKED	VECTIN

3.2.4 Control Flow and Structure Features

1. Control flow statements: BREAK, CYCLE, and RETURN
2. Sequential control operators: AND THEN and OR ELSE in IF, WHILE, REPEAT
3. Extended FOR loop: FOR VAR variable
4. VALUE section to initialize static variables
5. Mixed order LABEL, CONST, TYPE, VAR, VALUE sections
6. Compilable MODULES, with global attributes

7. UNIT INTERFACE and IMPLEMENTATION

- a. Interface version numbers, version checking
- b. Optional rename of constituents
- c. Guaranteed unique unit initialization
- d. Optional unit initialization

B.2.5 Extend-Level I/O and Files

1. Text file line length declaration, TEXT (nnn)
2. READ enumerated, Boolean, pointer, STRING, LSTRING
3. WRITE enumerated, pointer, LSTRING
4. Negative M value to justify left instead of right
5. Temporary files
6. DIRECT mode files, SEEK procedure
7. ASSIGN, CLOSE, DISCARD, READSET, READFN procedures
8. FILEMODES type and constants; F.MODE access
9. Error-trapping, F.TRAP and F.ERRS access
10. Enumerated I/O using identifier as string

B.2.6 System-Level I/O

The Wang PC Pascal extension to the ISO standard offers full FCBFQQ type equivalent to FILE types.

APPENDIX C

WANG PC PASCAL AND OTHER PASCALS

At the standard level, Wang PC Pascal conforms to the current ISO draft standard. In theory, therefore, programs written in accordance with the ISO standard are portable and can execute with the Wang PC Pascal compiler with no problem. In practice, however, the majority of Pascal programs have some nonstandard features. In these cases, it is necessary to alter the Pascal source file to conform to the conventions Wang PC Pascal uses.

C.1 IMPLEMENTATIONS OF PASCAL

The areas in which different implementations of the Pascal language differ from one another fall into the following categories:

1. Interactive I/O -- Wang PC Pascal implements lazy evaluation to help the compiler handle interactive I/O. Other versions of Pascal may implement this feature in different ways. For example, some systems require an initial READLN.
2. String-handling -- Wang PC Pascal supports the super array type LSTRING to handle variable-length strings efficiently. The ISO standard provides the PACK and UNPACK procedures for dealing with strings; other versions of Pascal often have some improvement on the string-handling facilities described in the standard.
3. Compiler controls -- Compiler controls that appear either as command line switches or as commands within source comments vary from Pascal to Pascal. To ensure portability, eliminate all embedded controls from comments.
4. Maximum set size -- The maximum set size varies from Pascal to Pascal. Some versions of Pascal limit set size to 16 or 64 elements. In Wang PC Pascal, sets can contain up to 256 elements. This allows support of the SET OF CHAR.
5. Type compatibility -- The rules for type compatibility vary in strictness. In some versions of Pascal, structurally equivalent types with different names are compatible; in others (and in the ISO Standard), they are not.
6. Out-of-block GOTOS -- Some versions of Pascal do not permit the out-of-block GOTOS that Wang PC Pascal does print.

Wang PC Pascal and Other Pascals

7. Heap management -- Rather than use the procedures NEW and DISPOSE for managing dynamic allocation of memory, some versions of Pascal use the MARK and RELEASE procedures. Wang PC Pascal supports both methods. (MARKAS and RELEAS are the Wang PC Pascal names for MARK and RELEASE.)
8. OTHERWISE in CASE statements and variant records -- If OTHERWISE is not present in a CASE statement, control does not automatically pass to the next executable statement as in some other versions of extended Pascal. Also, some other versions of Pascal use the word ELSE or OTHERS instead of OTHERWISE.
9. Assigning file names -- The ASSIGN procedure in Wang PC Pascal sets an operating system filename for a file. Some other versions of Pascal use a second parameter to RESET and REWRITE for the file name.
10. Separate compilation -- Most versions of Pascal exclude the EXTERN (or EXTERNAL) directive for procedures and functions. Many support the idea of a MODULE and/or an INTERFACE and IMPLEMENTATION, although the syntax may differ. Some do not support PUBLIC and EXTERN variables, but may use a FORTRAN COMMON approach. In the latter case, for portability, you should give all global variables in one Wang PC Pascal VAR section, using [PUBLIC] in the PROGRAM and [EXTERN] in the MODULE, and \$INCLUDE the same variable declarations in each.
11. Program parameters -- Some versions of Pascal ignore program parameters. In some versions of Pascal, all files must be program parameters.
12. Procedural parameters -- Several versions of Pascal do not permit passing procedures and functions as parameters. Many do not permit passing any predeclared procedures or functions.

C.2 WANG PC PASCAL AND UCSD PASCAL

Because UCSD Pascal is one of the more prevalent versions of Pascal for microcomputers, conversion of source files from UCSD to Wang PC Pascal, and vice versa, is likely to be a common occurrence. This section discusses the differences and similarities between the two versions of Pascal.

Wang PC Pascal has incorporated many of the UCSD extensions in one form or another. Table C-1 compares UCSD extensions with similar extensions available in Wang PC Pascal.

Table C-1. Wang PC Pascal and UCSD Pascal

UCSD Extension	Wang PC Pascal Equivalent
ATAN	ARCTAN
BLOCKREAD	GETUQQ
BLOCKWRITE	PUTUQQ
CLOSE	CLOSE
CLOSE (F, LOCK)	CLOSE (F)
CLOSE (F, PURGE)	DISCARD (F)
CONCAT	CONCAT
COPY	COPYLST or MOVEL
DELETE	DELETE
EXIT	RETURN or GOTO
FILLCHAR	FILLC and FILLSC
HALT	ENDXQQ
INSERT	INSERT
IORESULT, \$I	ERRS and TRAP fields
LENGTH	.LEN or STR [0]
LOG	LNDRQQ
MARK	MARKAS
MEMAVAIL	MEMAVL
MOVELEFT	MOVEL and MOVESL
MOVERIGHT	MOVER and MOVESR
POS	POSITN
RELEASE	RELEAS
SCAN	SCANEQ and SCANNE
SEEK	SEEK
SIZEOF	SIZEOF
STR	ENCODE
STRING [n]	LSTRING (n)
UNIT	UNIT
Untyped Files	FCBFQQ type

The following notes describe comparative points of interest.

1. The UCSD STRING [n] type is logically similar to the Wang PC Pascal LSTRING (n) type. Both contain the length of a variable length string in element zero of an ARRAY of CHAR.
2. UCSD Pascal allocates pointer variables on the heap with MARK and RELEASE. Other versions of Pascal normally use NEW and DISPOSE. Wang PC Pascal permits both methods of dynamic memory allocation.
3. Wang PC Pascal units are like UCSD Pascal units, with the following exceptions. In Wang PC Pascal, an INTERFACE must appear first in any compiland using it. Since UCSD Pascal has its own special file system, the name of the unit can find the interface file name in a standard way.

Wang PC Pascal requires a list of all identifiers exported from the unit in the UNIT clause itself. This list is optional in a USES clause. Different identifiers may appear in a USES clause to avoid identifier conflicts.

Finally, Wang PC Pascal provides for unit initialization code and interface version control. Neither of these are available in UCSD Pascal.

4. CONCAT is a function in UCSD Pascal; in Wang PC Pascal, it is a procedure.
5. In UCSD Pascal, when a CASE statement whose control value does not select another statement executes, the statement following the CASE statement also executes. In Wang PC Pascal, you must include an empty OTHERWISE clause to obtain this effect.
6. UCSD Pascal permits the use of the EOF (F) and EOLN (F) functions on a closed file; in Wang PC Pascal, this is an error.
7. UCSD Pascal permits comparison of records and arrays with the equal sign (=) and the not-equal sign (<>). In Wang PC Pascal, you must RETYPE the records and arrays to the same length STRING type, and then compare them as strings.

APPENDIX D
ASCII CHARACTER CODES

This appendix provides a list of ASCII character codes with their decimal and hexadecimal counterparts.

Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	032	20H	SPACE
001	01H	SOH	033	21H	!
002	02H	STX	034	22H	"
003	03H	ETX	035	23H	#
004	04H	EOT	036	24H	\$
005	05H	ENQ	037	25H	%
006	06H	ACK	038	26H	&
007	07H	BEL	039	27H	'
008	08H	BS	040	28H	(
009	09H	HT	041	29H)
010	0AH	LF	042	2AH	*
011	0BH	VT	043	2BH	+
012	0CH	FF	044	2CH	,
013	0DH	CR	045	2DH	-
014	0EH	SO	046	2EH	.
015	0FH	SI	047	2FH	/
016	10H	DLE	048	30H	0
017	11H	DC1	049	31H	1
018	12H	DC2	050	32H	2
019	13H	DC3	051	33H	3
020	14H	DC4	052	34H	4
021	15H	NAK	053	35H	5
022	16H	SYN	054	36H	6
023	17H	ETB	055	37H	7
024	18H	CAN	056	38H	8
025	19H	EM	057	39H	9
026	1AH	SUB	058	3AH	:
027	1BH	ESCAPE	059	3BH	;
028	1CH	FS	060	3CH	<
029	1DH	GS	061	3DH	=
030	1EH	RS	062	3EH	>
031	1FH	US	063	3FH	?

ASCII Character Codes

Dec	Hex	CHR	Dec	Hex	CHR
064	40H	@	096	60H	'
065	41H	A	097	61H	a
066	42H	B	098	62H	b
067	43H	C	099	63H	c
068	44H	D	100	64H	d
069	45H	E	101	65H	e
070	46H	F	102	66H	f
071	47H	G	103	67H	g
072	48H	H	104	68H	h
073	49H	I	105	69H	i
074	4AH	J	106	6AH	j
075	4BH	K	107	6BH	k
076	4CH	L	108	6CH	l
077	4DH	M	109	6DH	m
078	4EH	N	110	6EH	n
079	4FH	O	111	6FH	o
080	50H	P	112	70H	p
081	51H	Q	113	71H	q
082	52H	R	114	72H	r
083	53H	S	115	73H	s
084	54H	T	116	74H	t
085	55H	U	117	75H	u
086	56H	V	118	76H	v
087	57H	W	119	77H	w
088	58H	X	120	78H	x
089	59H	Y	121	79H	y
090	5AH	Z	122	7AH	z
091	5BH	[123	7BH	{
092	5CH	\	124	7CH	
093	5DH]	125	7DH	}
094	5EH	↑	126	7EH	!!
095	5FH	-	127	7FH	DEL

Dec=decimal, Hex=hexadecimal (H), CHR=character, LF=Line Feed, FF=Form Feed,
CR=Carriage Return, DEL=Rub out

APPENDIX E SUMMARY OF WANG PC PASCAL RESERVED WORDS

This appendix provides a list of the reserved words, names of attributes, and names of directives used in Wang PC Pascal.

The following are reserved words at the standard level:

AND	NIL
ARRAY	NOT
BEGIN	OF
CASE	OR
CONST	PACKED
DIV	PROCEDURE
DO	PROGRAM
DOWNT0	RECORD
ELSE	REPEAT
END	SET
FILE	THEN
FOR	TO
FUNCTION	TYPE
GOTO	UNTIL
IF	VAR
IN	WHILE
LABEL	WITH
MOD	

The following are reserved words at the extend-level:

BREAK	RETURN
CONSTS	SHL
CYCLE	SHR
IMPLEMENTATION	UNIT
INTERFACE	USES
ISR	VALUE
MODULE	VARS
OTHERWISE	XOR

The following are reserved words at the system level:

ADR	ADS
-----	-----

Summary of Wang PC Pascal Reserved Words

The following are names of attributes:

EXTERN	PORT
EXTERNAL	PUBLIC
FORTRAN	PURE
INTERRUPT	READONLY
ORIGIN	STATIC

The following are names of directives:

EXTERN
EXTERNAL
FORWARD

Logically, directives are reserved words. Since additional directives are legal in ISO Pascal, all are part of the standard level. EXTERN is both a directive and an attribute; EXTERNAL is a synonym for EXTERN in both cases, to provide compatibility with a number of other Pascals.

APPENDIX F
SUMMARY OF AVAILABLE PROCEDURES AND FUNCTIONS

Table F-1 provides an alphabetical list of all available functions and procedures, along with the group in which they are presented in Section 23.1.

Table F-1. Procedures and Functions

Name	Description	Category
ABORT	Terminate program	Extend-level
ABS	Absolute value function	Arithmetic
ACDRQQ	REAL8 arc cosine function	Arithmetic
ACSRQQ	REAL4 arc cosine function	Arithmetic
AIDRQQ	REAL8 truncate function	Arithmetic
AISRQQ	REAL4 truncate function	Arithmetic
ALLHQQ	Allocate heap item	Library
ANDRQQ	REAL8 round toward zero	Arithmetic
ANSRQQ	REAL4 round toward zero	Arithmetic
ARCTAN	Arc tangent function	Arithmetic
ASDRQQ	REAL8 arc sine function	Arithmetic
ASSRQQ	REAL4 arc sine function	Arithmetic
ASSIGN	Assign filename	File system
ATDRQQ	REAL8 arc tangent function	Arithmetic
ATSRQQ	REAL4 arc tangent (A/B)	Arithmetic
AZDRQQ	REAL8 arc tangent (A/B)	Arithmetic
AZSRQQ	REAL4 arc tangent function	Arithmetic
BEGQQQ	Initialize user	Library
BEGXQQ	Overall initialization	Library
BYLONG	WORD or INTEGER to INTEGER4	Extend-level
BYWORD	Put bytes in word	Extend-level
CHDRQQ	REAL8 hyperbolic cosine	Arithmetic
CHR	Get ASCII char of value	Data conversion
CHSRQQ	REAL4 hyperbolic cosine	Arithmetic
CLOSE	Close file	File system
CNDRQQ	REAL4 cosine function	Arithmetic
CNSRQQ	REAL4 cosine function	Arithmetic
CONCAT	Concatenate LSTRING	String
COPYLST	Copy to LSTRING	String
COPYSTR	Copy to STRING	String
COS	Cosine function	Arithmetic
DATE	Date function	Library
DECODE	Decode LSTRING to variable	Extend-level
DELETE	Remove portion of LSTRING	String
DISBIN	Disable interrupts	Library

Summary of Available Procedures and Functions

Table F-1. Procedures and Functions (continued)

Name	Description	Category
DISCARD	Close and delete file	File system
DISPOSE	Dispose of heap item	Dynamic allocation
ENABIN	Enable interrupts	Library
ENCODE	Encode expression to LSTRING	Extend-level
ENDQQQ	User termination	Library
ENDXQQ	Program termination	Library
EOF	Boolean end-of-file	File system
EOLN	Boolean end-of-line	File system
EVAL	Evaluate functions	Extend-level
EXDRQQ	REAL8 exponential function	Arithmetic
EXP	Exponential function	Arithmetic
EXSRQQ	REAL4 exponential function	Arithmetic
FILLC	Fill area with C, relative	System level
FILLSC	Fill area with C, segmented	System level
FLOAT	Convert INTEGER to REAL	Data conversion
FLOAT4	Convert INTEGER4 to REAL	Data conversion
FREETCT	Give count of free blocks	Library
GET	Get next file component	File system
GTUQQQ	Direct terminal input	Library
HIBYTE	Get high BYTE	Extend-level
HIWORD	Get high WORD	Extend-level
INSERT	Insert string	String
LADDOK	32-bit signed addition check	Library
LDDRQQ	REAL8 log base ten function	Arithmetic
LDSRQQ	REAL4 log base ten function	Arithmetic
LMULOK	32-bit signed multiply check	Arithmetic
LN	Natural log function	Arithmetic
LNRQQQ	REAL8 natural log	Arithmetic
LNSRQQ	REAL4 natural log	Arithmetic
LOBYTE	Get low BYTE	Extend-level
LOCKED	Resource locked status	Library
LOWER	Get lower bound	Extend-level
LOWORD	Get low WORD	Extend-level
MARKAS	Mark heap bounds	Library
MEMAVL	Available memory	Library
MDDRQQ	REAL8 modulo function	Arithmetic
MDSRQQ	REAL4 modulo function	Arithmetic
MNDRQQ	REAL8 minimum function	Arithmetic
MNSRQQ	REAL4 minimum function	Arithmetic
MOVEL	Move bytes left, relative	System level
MOVER	Move bytes right, relative	System level
MOVESL	Move bytes left, segmented	System level
MOVESR	Move bytes right, segmented	System level
MXDRQQ	REAL8 maximum function	Arithmetic
MXSRQQ	REAL4 maximum function	Arithmetic
NEW	Allocate new item heap	Dynamic allocation
ODD	Boolean odd function	Data conversion
ORD	Get ordinal value	Data conversion
PACK	Pack CHAR array	Data conversion
PAGE	Write new page	File System

Summary of Available Procedures and Functions

Table F-1. Procedures and Functions (continued)

Name	Description	Category
PIDRQQ	REAL8 to INTEGER power	Arithmetic
PISRQQ	REAL4 to INTEGER power	Arithmetic
PLYUQQ	Direct terminal end line	Library
POSITN	Find position of substring	String
PRED	Predecessor function	Data Conversion
PRDRQQ	REAL8 to REAL8 power	Arithmetic
PRSRQQ	REAL4 to REAL4 power	Arithmetic
PTYUQQ	Direct terminal output	Library
PUT	Put value to file	File system
READ	Read file	File system
READFN	Read file name	File system
READLN	Read file to end of line	File system
READSET	Read set	File system
RELEASE	Release heap space	Library
RETYPE	Force expression to type	System level
RESET	Ready file for read	File system
RESULT	Return result of function	Extend-level
REWRITE	Ready file for write	File system
ROUND	Round REAL	Data conversion
ROUND4	Round INTEGER4	Data conversion
SADDOK	16-bit signed addition check	Library
SCANEQ	Scan until char found	String
SCANNE	Scan until char not found	String
SEEK	Position at direct file record	File System
SHDRQQ	REAL8 hyperbolic sine	Arithmetic
SHSRQQ	REAL4 hyperbolic sine	Arithmetic
SIN	Sine function	Arithmetic
SIZEOF	Get size of structure	Extend-level
SMULOK	16-bit signed multiply check	Library
SQR	Square function	Arithmetic
SQRT	Square root function	Arithmetic
SUCC	Successor function	Data conversion
THDRQQ	REAL8 hyperbolic tangent	Arithmetic
THSRQQ	REAL4 hyperbolic tangent	Arithmetic
TICS	Time in arbitrary units	Library
TIME	Time of day function	Library
TNDRQQ	REAL8 tangent function	Arithmetic
TNSRQQ	REAL4 tangent function	Arithmetic
TRUNC	Truncate REAL	Data conversion
TRUNC4	Truncate INTEGER4	Data conversion
UADDOK	Unsigned addition check	Library
UMULOK	Unsigned multiply check	Library
UNLOCK	Unlock resource	Library
UNPACK	Unpack STRING to array	Data conversion
UPPER	Get upper bound	Extend-level
VECTIN	Set interrupt vector	Library
WRD	Convert to WORD file	Data conversion
WRITE	Write file	File system
WRITELN	Write line to file	File system

Summary of Available Procedures and Functions

APPENDIX G
SUMMARY OF WANG PC PASCAL METACOMMANDS

Table G-1 provides a single alphabetical list of all Wang PC Pascal metacommmands described in Chapter 26. The default listing for the metacommmand, if there is one, appears immediately after the name of the metacommmand.

Table G-1. Wang PC Pascal Metacommmands

Metacommmand	Action
\$BRAVE+	Sends messages to the screen
\$DEBUG-	Turns all error-checking on or off
\$ENTRY-	Generates procedure entry and exit calls for debugger
\$ERRORS:25	Sets number of errors allowed per page
\$EXTEND	Adds extend level features
\$GOTO-	Flags GOTOs as "considered harmful"
\$IF <constant> \$THEN <text1> \$ELSE <text2> \$END	Allows conditional compilation of <text1> source if <constant> is greater than zero
\$INCLUDE:'<file>	Switches compilation of file named
\$INCONST	Allows interactive setting of constant values during compilation
\$INDEXCK+	Checks for array index values in range
\$INITCK-	Checks for use of uninitialized values
\$INTEGER	Sets the length of the INTEGER type
\$LINE-	Generates line number calls for debugger
\$LINESIZE:79	Sets width of source listing

Summary of Wang PC Pascal Metacommmands

Table G-1. Wang PC Pascal Metacommands (continued)

Metacommand	Action
\$LIST+	Turns on or off source listing
\$MATHCK+	Checks for mathematical errors
\$NILCK+	Checks for bad pointer values
\$OCODE+	Turns on or off object code listing
\$PAGE+	Skips to next page
\$PAGE:n	Sets page number for next page
\$PAGEIF:n	Skips to next page if less than n lines left
\$PAGESIZE:55	Sets page length of source listing
\$POP	Restores saved value of all metacommands
\$PUSH	Saves current value of all metacommands
\$RANGECK+	Checks for subrange validity
\$REAL	Sets the length of the REAL type
\$ROM	Warns on static initialization
\$RUNTIME-	Determines context of runtime errors
\$SIMPLE	Disables global optimizations
\$SKIP:n	Skips n lines or to end of page
\$SPEED	Minimizes execution time of code
\$STACKCK+	Checks for stack overflow at entry
\$STANDARD	Enables standard level only
\$SUBTITLE:'<subt>'	Sets page subtitle
\$SYMTAB+	Sets symbol table to source listing
\$SYSTEM	Adds extend and system level features
\$TAGCK-	Checks tag fields in variant records
\$TITLE:'<title>'	Gives page title for source listing
\$WARN+	Gives warning messages in source listing

APPENDIX H

ERROR MESSAGES

This appendix lists all of the error numbers and messages you are likely to encounter while using the Wang PC Pascal compiler and runtime system. These error conditions fall into several categories.

1. Compiler warnings
2. Compiler errors caught
3. Compiler internal errors
4. Errors (both compiler and runtime) defined by the ISO standard that are not caught in Wang PC Pascal
5. Runtime file system errors
6. Runtime non-file system errors caught only if the appropriate switch is on
7. Runtime non-file system errors always caught

Linker errors are discussed in The Wang Professional Computer Program Development Guide.

Error conditions:

1. May go undetected.
2. May be detected by the compiler.
3. May be detected by the runtime system.

An error is "caught" if the compiler or runtime system detects the error and gives you a message. A "warning" is an error that the compiler catches but fixes so that the compiled source runs correctly.

Substitution mistakes (for example, using a colon instead of an equal sign) and some other syntax errors (for example, using a semicolon before an ELSE) are common errors that generate only a warning message; the compiler fixes them. You should, however, go back into the source file and make corrections, or you will get the same warning message every time you compile.

Compiler errors include all of the conditions described in this manual as "invalid", "illegal", "not permitted", and so on. The ISO standard defines a number of error conditions that are described as "errors not caught" in Wang PC Pascal. Usually, these are infrequent or very hard to detect conditions. They are not caught as errors in Wang PC Pascal, but might be considered so in another implementation.

H.1 COMPILER FRONT-END ERRORS

Front-end error and warning messages consist of a number and a message. Most messages appear with a row of dashes and an arrow that points to the location of the error; three (128, 129, and 130) appear only after the body of the routine in which they occur. The word "Warning" identifies warnings as such; all other messages report errors in the program.

The front end recovers from most errors; that is, it corrects the condition and continues the compilation. There are, however, a few front-end errors from which the compiler cannot recover. In these cases, you see the message:

Compiler Cannot Continue!

The compiler then does little else except list the rest of the program.

These errors occur under the following circumstances:

1. There are more errors than the number *n* you set with the \$ERRORS metacommand.
2. An end of file occurs unexpectedly.
3. Identifier scopes are nested too deeply.
4. The compiler cannot find the keyword PROGRAM, MODULE, or IMPLEMENTATION.
5. The compiler cannot find the PROGRAM, MODULE, or IMPLEMENTATION identifier.
6. A file system error occurs. The message includes the file name and one of the following phrases:

HARD DATA	for example, check sum error.
DISK FULL	Disk is full.
FILE ACCESS	for example, file not found.
FILE SYSTEM	Other or internal error.

The front end may also get one of two compiler runtime errors.

1. Error: Compiler Out of Memory

This usually occurs when you have declared too many identifiers. See Chapter 7 for suggestions on how to handle this situation.

2. Error: Compiler Internal Error

No matter what source program compiles, this message should not appear. If it does, please report the condition to the Wang Professional Computer Assistance Center (617-459-5000 in Massachusetts, Alaska, and Hawaii; 1-800-343-1098 elsewhere in the United States, Virgin Islands, or Puerto Rico).

If the word "Warning" appears before a message, the intermediate code files the front end produces are correct. The condition that produced the message is not severe, but is considered unsafe. Messages that indicate true errors halt any writing to intermediate files, which the compiler discards when the front end is finished.

The error message "Compiler" signifies the failure of an internal consistency check. No matter what source program compiles, this message should not appear. If it does, please report the condition to Wang at the number listed above.

The following list of compiler front-end errors includes the error number and message, with a brief explanation of the condition that generates the message.

101 Invalid Line Number

There are too many lines in the source file (limit is 32767).

102 Line Too Long Truncated

There are too many characters in the line (current limit is 142 characters).

103 Identifier Too Long Truncated

An identifier is longer than the maximum for your operating system and the compiler truncated it. See your Appendix A for the current maximum.

104 Number Too Long Truncated

A numeric constant is too long and the compiler truncated it. Numeric constants are limited to the same maximum length as identifiers.

105 End Of String Not Found

The line ended before the closing quotation mark appeared.

106 Assumed String

The compiler encountered double quotation marks (") or back-quotes and assumed that they enclose a string. Use single quotation marks instead.

107 Unexpected End Of File

While scanning, the compiler found an unexpected end-of-file in a number, metaccommand, or other illegal location.

108 Meta Command Expected Command Ignored

The compiler found a dollar sign (\$) at the start of a comment, but not a metaccommand identifier.

109 Unknown Meta Command Ignored

The compiler found a metaccommand identifier that it did not recognize or that was invalid in Wang PC Pascal

110 Constant Identifier Unknown Or Invalid Assumed Zero

The constant identifier following a metaccommand is unknown (as in \$DEBUG: A) or not a constant of the right type. The compiler has replaced the unknown or incorrect value with zero.

112 Invalid Numeric Constant Assumed Zero

The constant following a metaccommand was a numeric constant (for example, \$DEBUG: 123456) that had the wrong format or was out of range. The compiler has replaced the incorrect value with zero.

113 Invalid Meta Value Assumed Zero

The value following a metaccommand was neither a constant nor an identifier. The compiler has replaced the incorrect value with zero.

114 Invalid Meta Command

The compiler expected but did not find one of the following after a metaccommand: +, -, or :. The compiler has ignored the metaccommand.

116 Meta Value Out Of Range Skipped

The integer value given for the \$LINESIZE metaccommand was below 16 or above 160. Or, n was not either 4 or 8 for \$REAL:n or 2 for \$INTEGER. In any of these cases, the compiler ignores the metaccommand.

- 117 File Identifier Too Long Skipped
- The string value given for the file name in a \$INCLUDE metaccommand was too long. The compiler ignored the metaccommand. The maximum is 96 characters.
- 118 Too Many File Levels
- Too many nested levels of files were brought in by the \$INCLUDE metaccommand. The \$INCLUDE metaccommand has been ignored.
- 119 Invalid Initialize Meta
- A \$POP metaccommand has no corresponding \$PUSH metaccommand.
- 120 CONST Identifier Expected
- The compiler did not find an identifier following an \$INCONST metaccommand. The \$INCONST metaccommand has been ignored.
- 121 Invalid INPUT Number Assumed Zero
- The user input invoked by \$INCONST was invalid in some way; the compiler assumes to be zero.
- 122 Invalid Meta Command Skipped
- The compiler found an \$IF metaccommand but no subsequent \$THEN or \$ELSE. The compiler thus ignored the \$IF command.
- 123 Unexpected Metaccommand Skipped
- The compiler found a metaccommand not enclosed in comment delimiters, but processed it anyway.
- 126 Invalid Real Constant
- The compiler found a type REAL constant with a leading or a trailing decimal point. But the constant's value was accepted anyway.
- 127 Invalid Character Skipped
- The compiler found a character in the source file that was not acceptable in program text.
- 128 Forward Proc Missing: <procedure>
- The compiler found a procedure or function declared FORWARD but could not find the procedure or function itself. This message appears in the symbol table area of the listing file.

129 Label Not <label>

The compiler could not find any use of a label you declared in a LABEL section. This message occurs in the symbol table area of the listing file.

130 Program Parameter Bad: <parameter>

The compiler encountered this program parameter, which you never declared or has an unacceptable type. This message occurs in the symbol table area of the listing file.

133 Type Size Overflow

The data type declared implies a structure bigger than the maximum of 65534 bytes.

134 Constant Memory Overflow

Constant memory allocation has exceeded the maximum of 65534 bytes.

135 Status Memory Overflow

Static memory allocation has exceeded the maximum of 65534 bytes.

136 Stack Memory Overflow

Stack frame memory allocation has exceeded the maximum of 65534 bytes.

137 Integer Constant Overflow

The value of a type INTEGER, signed constant expression is out of range.

138 Word Constant Overflow

The value of a type WORD or other unsigned constant expression is out of range.

139 Value Not In Range For Record

In a structured constant, long form of the NEW, DISPOSE, or SIZEOF procedure, or other application, the record tag value is not in the range of the variant.

140 Too Many Compiler Labels

The compiler needs internal labels, and the program is too big. You must break your program into smaller pieces.

141 Compiler

142 Too Many Identifier Levels

The identifier scope level exceeds 15.

143 Compiler

144 Compiler

This error may occur if the PASKEY file format is incorrect.

145 Identifier Already Declared

The compiler found an identifier declared more than once in a given scope level.

146 Unexpected End Of File

While parsing, the compiler found an end-of-file where it should not be in a statement, declaration, etc.

147 : Assumed =

The compiler found a colon where there should have been an equals sign and proceeded as if the correct symbol were present.

148 = Assumed :

The compiler found an equals sign where it expected a colon and proceeded as if the correct symbol were present.

149 := Assumed =

The compiler found colon followed by an equals sign where it expected an equals sign only and proceeded as if the correct symbol were present.

150 = Assumed :=

The compiler found an equals sign where it expected a colon followed by an equals sign and proceeded as if the correct symbol were present.

151 { Assumed (

The compiler found a left bracket where it expected a left parenthesis and proceeded as if the correct symbol were present.

152 (Assumed [

The compiler found a left parenthesis where it expected a left bracket and proceeded as if the correct symbol were present.

153) Assumed]

The compiler found a right parenthesis where it expected a right bracket and proceeded as if the correct symbol were present.

154] Assumed)

The compiler found a right bracket where it expected a right parenthesis and proceeded as if the correct symbol were present.

155 ; Assumed ,

The compiler found a semicolon where it expected a comma and proceeded as if the correct symbol were present.

156 , Assumed ;

The compiler found a comma where it expected a semicolon and proceeded as if the correct symbol were present.

162 Insert Symbol

The compiler did not find a symbol it expected, but proceeded as if it were present. This message should not occur; it is a minor compiler error. If it does, contact the Wang Professional Computer Assistance Center at the number listed on Page H-3.

163 Insert ,

The compiler did not find a comma where it expected one, but proceeded as if it were present.

164 Insert ;

The compiler did not find a semicolon where it expected one, but proceeded as if it were present.

165 Insert =

The compiler did not find an equals sign where it expected one, but proceeded as if it were present.

166 Insert :=

The compiler did not find a colon followed by an equals sign where it expected one, but proceeded as if it were present.

167 Insert OF

The compiler did not find an OF where it expected one, but proceeded as if it were present.

168 Insert]

The compiler did not find a right bracket where it expected one, but proceeded as if it were present.

- 169 Insert)
The compiler did not find a right parenthesis where it expected one, but proceeded as if it were present.
- 170 Insert {
The compiler did not find a left bracket where it expected one, but proceeded as if it were present.
- 171 Insert (
The compiler did not find a left parenthesis where it expected one, but proceeded as if it were present.
- 172 Insert DO
The compiler did not find a DO where it expected one, but proceeded as if it were present.
- 173 Insert :
The compiler did not find a colon where it expected one, but proceeded as if it were present.
- 174 Insert .
The compiler did not find a period where it expected one, but proceeded as if it were present.
- 175 Insert ..
The compiler did not find a double period where it expected one, but proceeded as if it were present.
- 176 Insert END
The compiler did not find an END where it expected one, but proceeded as if it were present.
- 177 Insert TO
The compiler did not find a TO where it expected one, but proceeded as if it were present.
- 178 Insert THEN
The compiler did not find a THEN where it expected one, but proceeded as if it were present.
- 179 Insert *
The compiler did not find an asterisk where it expected one, but proceeded as if it were present.

Error Messages

185 Invalid Symbol Begin Skip

186 End Skip

The compiler found a symbol it expected, but only after some other invalid symbols. The compiler skipped the invalid symbols, beginning at the point where message #185 appears and ending where message #186 appears.

187 End Skip

This message marks the end of skipped source text for any message, except #185, that ended with the phrase "Begin Skip."

188 Section Or Expression Too Long

The compiler has reached its limit. Try rearranging the program or breaking up an expression with assignments to intermediate values.

189 Invalid Set Operator Or Function

Your source file includes an incorrect use of a set operator or function (for example, MOD operator or ODD function with sets).

190 Invalid Real Operator Or Function

Your source file includes an incorrect use of an operator or function on a REAL value (for example, MOD operator or ODD function with reals).

191 Invalid Value Type For Operator Or Function

For example, MOD operator or ODD function with enumerated type.

195 Compiler

196 Zero Size Value

Your source file includes the empty record "RECORD END" as if it has a size.

197 Compiler

198 Constant Expression Value Out Of Range

The value of a constant expression is out of range in an array index, subrange assignment, or other subrange.

199 Integer Type Not Compatible With Word Type

An expression tries to mix INTEGER and WORD type values. This common error indicates confusing signed and unsigned arithmetic; either change the positive signed value to unsigned with WRD () or change the unsigned value (<MAXINT) to signed with ORD().

- 201 Types Not Assignment Compatible
- You have attempted to use incompatible types in an assignment statement or value parameter. See Chapter 13 in the manual for type compatibility rules.
- 202 Types Not Compatible In Expression
- You have attempted to mix incompatible types in an expression. See Chapter 13 in this manual for type compatibility rules.
- 203 Not Array Begin Skip
- A variable followed by a left bracket (or parenthesis) is not array. The compiler has skipped from here to where message 187 appears.
- 204 Invalid Ordinal Expression Assumed Integer Zero
- The expression has the wrong type or a type that is not ordinal. The compiler assumes the value of the expression to be zero.
- 205 Invalid Use Of PACKED Components
- A component of a PACKED structure has no address (it may not be on a byte boundary) and cannot be passed by reference.
- 206 Not Record Field Ignored
- A variable followed by a period is not a record, address, or file, and the compiler has ignored it.
- 207 Invalid Field
- A valid field name does not follow a record variable and a period, and the compiler has ignored it.
- 208 File Dereference Considered Harmful
- When the compiler calculates the address of a file buffer variables, it cannot do the special actions normally done with buffer variables (for example, lazy evaluation, for text files, or concurrency, for binary files). Since the buffer variable at this address may not be valid, such a practice is considered harmful.
- 209 Cannot Dereference Value
- The variable followed by an arrow is not a pointer, address, or file; therefore the compiler cannot dereference the value pointed to.
- 210 Invalid Segment Address
- A variable resides at segmented address, but a default segment address is needed. You may need to make a local copy of the variable.

211 Ordinal Expression Invalid Or Not Constant

The compiler found an invalid or non-constant expression where it expected a constant ordinal expression.

214 Out Of Range For Set 255 Assumed

The compiler found an element of a set constant whose ordinal value exceeded 255 and assumed a value of 255.

215 Type Too Long Or Contains File Begin Skip

The compiler found a structured constant that exceeded 255 bytes or either is or contains a FILE or LSTRING type.

216 Extra Array Components Ignored

The compiler found an array constant that had too many components for the array type. The compiler ignored excess components.

217 Extra Record Components Ignored

The compiler found a record constant that had too many components for the record type. The compiler ignored excess components.

218 Constant Value Expected Zero Assumed

The compiler found a nonconstant value in a structured constant and assumed its value was zero.

220 Compiler

221 Components Expected For Type

The compiler found too few components for the type of a structured constant.

222 Overflow 255 Components In String Constant

The compiler found a string constant that exceeded 255 bytes.

223 Use NULL

Use the predeclared constant NULL instead of two quotation marks.

224 Cannot Assign With Supertype Lstring

A super array LSTRING cannot be the source or the target of an assignment.

225 String Expression Not Constant

String concatenation with the asterisk applies only to constants.

226 String Expected Character 255 Assumed

The compiler found a string constant with no characters, perhaps the result of using NULL, and assumed the value CHR(255).

227 Invalid Address Of Function

An assignment or other address reference to the function value is not within the scope of the function. Or, you used RESULT outside the scope of the function.

228 Cannot Assign To Variable

Assignment to READONLY, CONST, or FOR control variable is not permitted.

230 Unknown Identifier Assumed Integer Begin Skip

The compiler found an unknown identifier, for which it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

231 VAR Parameter Or WITH Record Assumed Integer Begin Skip

The compiler found an invalid symbol where it requires an address, and has skipped to a comma, semicolon, or right parenthesis.

232 Cannot Assign To Type

The target of an assignment is a file or the compiler cannot assign it for some other reason.

233 Invalid Procedure Or Function Parameter Begin Skip

The compiler found an incorrect use of an intrinsic procedure or function. The error could be one of the following:

1. The first parameter of NEW or DISPOSE is not a pointer variable.
2. The record tag value of a NEW, DISPOSE, or SIZEOF procedure could not be found.
3. The super array in a NEW, DISPOSE, or SIZEOF procedure had too many bounds.
4. The super array in a NEW, DISPOSE, or SIZEOF procedures had too few bounds.
5. The super array for a NEW or SIZEOF procedure has been given no bounds.
6. You attempted to use the ORD or WRD function on a value not of an ordinal type.
7. You attempted to use the LOWER or UPPER functions on an invalid or type.

8. PACK or UNPACKA on super array or file, or an array that is or is not packed as expected.
9. The first parameter for a RETYPE is not a type identifier.
10. The parameter for a RESULT function is not a function identifier.
11. You attempted to use an intrinsic procedure or function not available in this version of Wang PC Pascal.
12. The ORD or WRD of an INTEGER4 value is out of range.
13. The parameter given for HIWORD or LOWORD is not an ordinal or INTEGER4.

234 Type Invalid Assumed Integer

The parameter you specified in READ, WRITE, ENCODE, or DECODE is not of type INTEGER, WORD, INTEGER4, REAL, BOOLEAN, enumerated, a pointer; or, the parameter given for a READ or WRITE is not of type CHAR, STRING, LSTRING; or, the parameter for a READFN is not of one of these types or type FILE. The compiler has assumed it to be of type INTEGER. This error also occurs if a program parameter does not have a readable type, in which case the error occurs at the keyword BEGIN for the main program.

235 Assumed File INPUT

Because the first parameter for a READFN is not a file, the compiler assumes INPUT.

236 Invalid Segment For File

File parameters must always reside in the default segment.

237 Assumed INPUT

INPUT was not given as a program parameter and the compiler assumes it is present.

238 Assumed OUTPUT

OUTPUT was not given as a program parameter and the compiler assumes it is present.

239 Not Lstring Or Invalid Segment

The target of a READSET, ENCODE, or DCODE must be an LSTRING in the default segment. One or both of these conditions is missing.

242 File Parameter Expected Begin Skip

The READSET procedure expects, but cannot find, a text file parameter. The compiler has ignored the procedure and resume where message 187 appears.

- 243 Character Set Expected
- The READSET procedure expects, but cannot find, a SET OF CHAR parameter.
- 244 Unexpected Parameter Begin Skip
- The compiler found more than one parameter given for an EOF, EOLN, or PAGE, and has ignored the extra.
- 245 Not Text File
- You attempted to use an EOLN, PAGE, READLN, or WRITELN on some file other than a text file.
- 248 Size Not Identical
- The RETYPE function may not work as intended, since the parameters you specified are of unequal length.
- 249 Procedural Type Parameter List Not Compatible
- The parameter lists for formal and actual procedural parameters are not compatible. That is, the number of parameters, the function result type, a parameter type, or attributes are different.
- 250 Cannot Use Procedure With Attribute
- You attempted to call a procedure with the attribute INTERRUPT, directly or indirectly. INTERRUPT does not allow this.
- 251 Unexpected Parameter Begin Skip
- The compiler found a left parenthesis, indicating a procedure or function, but no parameters and has skipped to where message 187 appears.
- 252 Cannot Use Procedure Or Function As Parameter
- You attempted to pass this intrinsic procedure or function as a parameter, which is illegal.
- 253 Parameter Not Procedure Or Function Begin Skip
- The compiler expected, but cannot find, a procedural parameter here, and has skipped to where message 187 appears.
- 254 Supertype Array Parameter Not Compatible
- The actual parameter given is not of the same type or is not derived from the same super type as the formal parameter.
- 255 Compiler

256 VAR or CONST Parameter Types Not Identical

The actual and formal reference parameter types are not identical, as they must be.

257 Parameter List Size Wrong Begin Skip

The compiler found too many or too few parameters in a list. If too many, the compiler skipped the excess ones.

258 Invalid Procedural Parameter to EXTERN

A procedure or function that is neither PUBLIC nor EXTERN is being passed as a parameter to a procedure or function declared EXTERN. (The compiler invokes the actual procedure or function with intrasegment calls, and so cannot pass them to an external code segment.)

259 Invalid Set Constant For Type

The set is not constant, base types are not identical, or the constant is too big.

260 Unknown Identifier In Expression Assumed Zero

The identifier in an expression is undefined or possibly misspelled.

261 Identifier Wrong In Expression Assumed Zero

The identifier in an expression is incorrect (for example., file type ID) and the compiler assumes it to be zero.

262 Assumed Parameter Index Or Field Begin Skip

After error 260 or 261, anything in parentheses or square brackets, or a dot followed by an identifier, is skipped.

265 Invalid Numeric Constant Assumed Zero

There is a decode error in an assumed INTEGER or INTEGER4 literal constant; the number is too big, has invalid characters, etc. The compiler assumes the incorrect constant to be zero.

267 Invalid Real Numeric Constant

There is a decode error in an assumed type REAL literal constant; the number is too big, has invalid characters, etc.

268 Cannot Begin Expression Skipped

The compiler deleted a symbol that cannot start an expression.

269 Cannot Begin Expression Assumed Zero

The compiler deleted a symbol that cannot start an expression with a zero.

270 Constant Overflow

The divisor in a DIV or MOD function is the constant zero (INTEGER or WORD), which is illegal.

272 Word Constant Overflow

A WORD constant minus a WORD constant has given a negative result.

273 Invalid Range

The lower bound of a subrange is greater than the upper bound (for example, 2..1).

276 CASE Constant Expected

The compiler expects, but cannot find, a constant value for a CASE statement or record variant.

277 Value Already In Use

In a CASE statement or record variant, the value has already been assigned (as in CASE 1..3: XX; 2: YYY; END).

279 Label Expected

The compiler expects, but cannot find, a label.

280 Invalid Integer Label

A label uses nondecimal notation (for example, 8#77), which is illegal.

281 Label Assumed Declared

The compiler found a label that did not appear in the LABEL section.

283 Expression Not Boolean Type

The expression following an IF, WHILE, or UNTIL statement must be BOOLEAN.

284 Skip To End Of Statement

The compiler found, and has skipped, an unexpected ELSE or UNTIL clause.

285 Compiler

286 ; Ignored

The compiler found, and has ignored, a semicolon before an ELSE statement. (The semicolon is not required in this case.)

288 : Skipped

The compiler found, and has ignored, a colon after an OTHERWISE statement. (The colon is not required in this case.)

289 Variable Expected For FOR Statement Begin Skip

The compiler expects, but cannot find, a variable identifier after a FOR statement and has skipped to where message #187 appears.

291 FOR Variable Not Ordinal Or Static Or Declared In Procedure

The compiler has found an incorrect control variable in a FOR statement. Specifically, the control variable is, but should not be, one of the following:

1. Type REAL, INTEGER4, or another non-ordinal type
2. The component of an array, record, or file type
3. The referent of a pointer type or address type
4. In the stack or heap, unless locally declared
5. Nonlocally declared, unless in static memory
6. A reference parameter (VAR or VARS parameter)
7. A variable with a segmented ORIGIN attribute

292 Skip To :=

The compiler expects, but cannot find, an assignment in a FOR statement, and has skipped to the next :=.

293 GOTO Invalid

The GOTO or label here involves an invalid GOTO statement.

294 GOTO Considered Harmful

As directed, if the \$GOTO metaccommand is on, the compiler has found a GOTO statement.

296 Label Not Loop Label

The label after a BREAK or CYCLE statement is not a loop label (for example, it does not label a FOR, WHILE, or REPEAT statement).

297 Not In Loop

The compiler had found a BREAK or CYCLE statement outside a FOR, WHILE, or REPEAT statement.

298 Record Expected Begin Skip

The compiler expects, but cannot find, a record variable in a WITH statement and has skipped to where message 187 appears.

300 Label Already In Use Previous Use Ignored

The compiler found a label that has already appeared in front of a statement and has ignored the previous use.

301 Invalid Use Of Procedure Or Function Parameter

The compiler has found a procedure parameter used as a function or a function parameter used as a procedure.

303 Unknown Identifier Skip Statement

The compiler has found an undefined (or possibly misspelled) identifier at the beginning of a statement and has ignored the entire statement.

304 Invalid Identifier Skip Statement

The compiler has found an incorrect identifier at the beginning of a statement (for example, file type id) and has ignored the entire statement.

305 Statement Not Expected

The compiler has found a MODULE or uninitialized IMPLEMENTATION with a body enclosed with the reserved words BEGIN and END.

306 Function Assignment Not Found

The compiler expects, but cannot find, an assignment of the value of a function somewhere in its body.

307 Unexpected END Skipped

The compiler found, and ignored, an END without a matching BEGIN, CASE or RECORD.

308 Compiler**309 Attributed Invalid**

The compiler found an attribute valid only for procedures and functions given to a variable, an attribute valid only for a variable given to a procedure or function, or an invalid mix of attributes (for example, PUBLIC and EXTERN).

310 Attribute Expected

The compiler expects, but cannot find, a valid attribute, following the left bracket.

311 Skip To Identifier

The compiler skipped an invalid (for example, unexpected) symbol to get to the identifier that follows.

312 Identifier Expected

The compiler found something not an identifier where it expected a list of identifiers.

314 Identifier Expected Skip To ;

The compiler expects, but cannot find, the declaration of a new identifier and has skipped to the next semicolon.

315 Type Unknown Or Invalid Assumed Integer Begin Skip

The return type for a parameter or function is incorrect; that is, it is not an identifier or is undeclared, or the value parameter or function return is a file or super array. The compiler has assumed the type if INTEGER and skipped to where message 187 appears.

316 Identifier Expected

The compiler expects, but cannot find, an identifier after the word PROCEDURE or FUNCTION in parameter list.

318 Compiler

319 Compiler

320 Previous Forward Skip Parameter List

The compiler found a definition of a FORWARD (or INTERFACE) procedure or function that unnecessarily repeats the parameter list and function return type.

321 Not EXTERN

The compiler found a procedure or function with the ORIGIN attribute but lacking the EXTERN attribute as well.

322 Invalid Attribute With Function Or Parameter

The compiler found an incorrectly-used INTERRUPT procedure, that is, one that has parameters or is a function.

323 Invalid Attribute In Procedure Or Function

The compiler has found a nested procedure or function that has attributes or is declared EXTERN. Neither of these conditions is legal.

324 Compiler

- 325 **Already Forward**
- You attempted to use **FORWARD** twice for the same procedure or function.
- 326 **Identifier Expected For Procedure Or Function**
- The compiler expects, but cannot find, an identifier following the keywords **PROCEDURE** or **FUNCTION**.
- 327 **Invalid Symbol Skipped**
- The compiler found, and ignored, a **FORWARD** or **EXTERN** directive in an interface.
- 328 **EXTERN Invalid With Attribute**
- The compiler found an **EXTERN** procedure also declared **PUBLIC**. This is illegal.
- 329 **Ordinal Type Identifier Expected Integer Assumed Begin Skip**
- The compiler expects, but cannot find, an ordinal type identifier for a record tag type. It has skipped what is in the source file and assumed type **INTEGER**.
- 330 **Contains File Cannot Initialize**
- You have used a file in a record variant. This is legal, but considered unsafe, and the compiler does not initialize it automatically with the usual **NEWFQQ** call.
- 331 **Type Identifier Expected Assumed Character**
- The compiler expects, but cannot find, an ordinal type identifier. It assumes that what it does find is of type **CHAR**.
- 333 **Not Supertype Assumed String**
- The compiler has found what looks like a super array type designator. However, the type identifier is not for a super array type, so the compiler assumes it to be of the super array type **STRING**.
- 334 **Type Expected Integer Assumed**
- The compiler expects, but cannot find, a type clause or type identifier and has assumed the expected type to be type **INTEGER**.
- 335 **Out Of Range 255 For Lstring**
- The compiler has found an **LSTRING** designator whose upper bound exceeds 255.

336 Cannot Use Supertype Use Designator

A super array type can only be used as a reference parameter or a pointer referent. You cannot give other variables a super array type. Use a super array designator.

337 Supertype Designator Not Found

The compiler expects, but cannot find, a super array designator that gives the upper bounds of the super array.

338 Contains File Cannot Initialize

The compiler has found a super array of a file type. While allowed, this is considered unsafe and is not initialized automatically with the usual NEWFQQ call.

339 Supertype Not Array Skip To ; Assumed Integer

The compiler expects, but cannot find, the keyword ARRAY following SUPER in a type clause. It has assumed that the type is INTEGER and skipped to the next semicolon.

340 Invalid Set Range Integer Zero To 255 Assumed

The compiler has found an invalid range for the base type of a set and assumed it to be of type INTEGER with a range from zero to 255.

341 File Contains File

The compiler has found, but does not permit, a file type that contains a file type, either directly or indirectly.

342 PACKED Identifier Invalid Ignored

The compiler expects, but cannot find, one of words ARRAY, RECORD, SET, or FILE following the reserved word PACKED. Any type identifier following PACKED is illegal.

343 Unexpected PACKED

The compiler found the keyword PACKED applied to one of the nonstructured types.

345 Skip To ;

The compiler expects, but cannot find, a semicolon at the end of a declaration (which is not at the end of the line). It has assumed the next semicolon is the end of the declaration.

- 346 Insert ;
- The compiler expects, but cannot find, a semicolon at the end of a declaration (which coincides with the end of the line). It has inserted a semicolon where it expected to find one.
- 347 Cannot Use Value Section With ROM Memory
- If the \$ROM metaccommand is on, you cannot also have a VALUE section.
- 348 UNIT Procedure Or Function Invalid EXTERN
- A required EXTERN declaration occurs later than it should in an IMPLEMENTATION. (You must declare any interface procedures and functions not implemented EXTERN at the beginning.)
- 350 Not Array Begin Skip
- The variable followed by a left bracket, in a VALUE section, is not an array.
- 351 Not Record Begin Skip
- The variable followed by a period, in VALUE section, is not a record type.
- 352 Invalid Field
- Within a VALUE section, the identifier the compiler assumes to be a field is not in the record.
- 353 Constant Value Expected
- Within a VALUE section, you have initialized a variable to something other than a constant.
- 354 Not Assignment Operator Skip To ;
- Within a VALUE section, the assignment operator is missing.
- 355 Cannot Initialize Identifier Skip To ;
- Within a VALUE section, there is a symbol that is not a variable you declared at this level in fixed (STATIC) memory. Or, it has an illegal ORIGIN or EXTERN attribute.
- 356 Cannot Use Value Section
- A VALUE section has been incorrectly included in the INTERFACE, rather than in the IMPLEMENTATION.

357 Unknown Forward Pointer Type Assumed Integer

The identifier for the referent of a reference type declared earlier in this TYPE (or VAR) section was never declared itself.

358 Pointer Type Assumed Forward

The TYPE section includes a pointer or address type for which the referent type was already declared in an enclosing scope. Since the identifier for the referent type was declared again later in the same TYPE section, the compiler used the second definition. In the following example the forward type, REAL, IS USED:

```
PROGRAM OUTSIDE;
TYPE A = WORD;
  PROCEDURE B;
    TYPE C= ^A;
    A = REAL;
```

359 Cannot Use Label Section

The compiler found a LABEL section incorrectly included in an INTERFACE, rather than in an IMPLEMENTATION.

360 Forward Pointer To Supertype

The referent of a reference type declared in the TYPE section is a super array type. The declaration the super array type does not occur until after the reference.

361 Constant Expression Expected Zero Assumed

An expression is a CONST section is not constant.

362 Attribute Invalid

A VAR section mixes incorrectly the PUBLIC or ORIGIN attribute with EXTERN. Or, ORIGIN appears in attribute brackets after the keyword VAR.

364 Contains File Initialize Module

The compiler found an uninitialized file variable in a module. You must call the module as a parameterless procedure to initialize the files.

365 Origin Variable Contains File Cannot Initialize

The compiler found an uninitialized file variables with the ORIGIN attribute. Since ORIGIN variables are never initialized, you must initialize this file yourself.

366 UNIT Identifier Expected Skip To ;

The compiler expects, but cannot find, an identifier after the keyword USES.

- 367 **Initialize Module To Initialize UNIT**
- You must call the module as a procedure in order to initialize it (a USES clause triggers an unit initialization call).
- 368 **Identifier List Too Long Extra Assumed Integer**
- In a USES clause with a list of identifiers, the compiler found more identifiers in the list than are constituents of the interface. The extra ones are assumed to be type identifiers identical to INTEGER.
- 369 **End of UNIT Identifier List Ignored**
- In a USES clause with a list of identifiers, the compiler found fewer identifiers in the list than are constituents of the interface. The remaining interface constituents are not part of the USES clause.
- 371 **UNIT Identifier Expected**
- An identifier is missing after the phrase "INTERFACE; UNIT".
- 372 **Compiler**
- Compiler expects, but cannot find, the keyword UNIT in an INTERFACE
- 373 **Identifier In UNIT List Not Declared**
- One of the identifiers in the interface UNIT list is not in the body of the interface.
- 374 **Program Identifier Expected**
- An identifier is missing after the keyword PROGRAM or MODULE.
- 375 **UNIT Identifier Expected**
- The unit identifier is missing after the phrase "IMPLEMENTATION OF".
- 376 **Program Not Found**
- The compiler expects, but cannot find, one of the reserved words PROGRAM, MODULE, or IMPLEMENTATION OF. (This error can occur if the source file is not a Pascal compiland.)
- 377 **File End Expected Skip To End**
- The compiler found addition source text after what appeared to be the end and ignored everything after what it thought was the end.
- 378 **Program Not Found**
- The compiler expects, but cannot find, the main body of a compiland or the final END.

H.2 COMPILER BACK-END ERRORS

The main source of back-end errors is user errors from either the optimizer or the code generator. There are, in fact, very few of these errors exist. All are concerned with limitations that the front end cannot detect.

Back-end errors cause an immediate abort, while an error number and approximate listing line number appear on your screen. Back-end errors include:

1. Attempt to divide by zero.

For example, A DIV 0.

2. Overflow during integer constant folding.

For example, MAXINT + A + MAXINT.

3. Expression too complex/Too many internal labels.

Try breaking up expression with intermediate value assigns.

4. Too many procedures and/or functions [Pcode only].

5. Range error (number too large to fit into target).

H.3. COMPILER INTERNAL ERRORS

All errors labeled "Compiler" in Section H.1 are compiler internal errors that should never occur. If one does occur, report it to the Wang Professional Computer Assistance Center at the number listed on Page H-3.

The back end of the compiler also makes many internal consistency checks. These checks should always be correct and never give an internal error.

When they do occur, back-end internal error messages have the following format:

*** Internal Error NNN

NNN is the internal error number, which ranges from 1 to 999. There is little you can do when an internal error occurs, except report it and perhaps modify your program near the line where the error occurred.

H.4 RUNTIME FILE SYSTEM ERRORS

File system error codes range from 1000 to 1999. Error codes go into the ERRC field of the file control block. Codes from 1000 to 1099 designate errors (from Unit U) that are specific to your operating system. Those from 1100 to 1199 identify Pascal file system errors (from unit F).

File system errors all have the format:

<error type> error in file <filename>

followed by the error code, and in some versions an error status, which is an operating system error return word. The <error type> field is based on the ERRS field of the file control block, as follows:

1. Hard Data
Hard data error (parity, CRC, checksum, etc.)
2. Device Name
Invalid unit/device/volume name format or number.
3. Operation
Invalid operation: GET if OEF, RESET a printer, etc.
4. File System
File system internal error, ERRS >15, etc.
5. Device Offline
Unit/device/volume no longer available.
6. Lost File
File itself no longer available.
7. File Name
Invalid syntax, name too long, no temp names, etc.
8. Device Full
Disk full, directory full, all channels allocated.
9. Unknown Device
Unit/device/volume not found.

10. File Not Found

File itself not found.

11. Protected File

Duplicate filename; write-protected

12. File In Use

File in use, concurrency lock, already open.

13. File Not Open

File closed, I/O to unopen FCB.

14. Data Format

Data format error, decode error, range error.

15. Line Too Long

Buffer overflow, line too long.

H.4.1 Operating System Runtime Errors

The following error messages are specific to particular operating systems.

1000 Write error when writing end of file

1001 Unknown device name

CP/M-80 and CP/M-86 only. Occurs when you have not assigned a file name in a RESET or REWRITE.

1002 File name extension with more than 3 characters

1003 Error during creation of new file

(disk or directory full)

1004 Error during open of existing file (file not found)

1005 File name with more than 8 or zero characters

1006 Device cannot do input or output

(CP/M-80 and CP/M-86 only)

1007 Total file name length over 21 characters

1008 Write error when advancing to next record

Error Messages

- 1009 File too big (over 65535 logical sectors)
- 1010 Write error when seeking to direct record
- 1011 Attempt to pen a random file to a non-disk device
- 1012 Forward space or back space on a non-disk device (FORTRAN error only)
- 1013 Disk or directory full error during forward space or back space (FORTRAN error only)

H.4.2 Wang PC Pascal File System Error Codes (100-1199)

- 1100 ASSIGN or READFN of file name to open file
- 1101 Reference to buffer variable of closed textfile
- 1102 Textfile READ or WRITE call to closed file
- 1103 READ when EOF is true (SEQUENTIAL mode)
- 1104 READ to REWRITE file, or WRITE to RESET file (SEQUENTIAL mode)
- 1105 EOF call to closed file
- 1106 GET call to closed file
- 1107 GET call when EOF is true (SEQUENTIAL mode)
- 1108 GET call to REWRITE file (SEQUENTIAL mode)
- 1109 PUT call to closed file
- 1110 PUT call to RESET file (SEQUENTIAL mode)
- 1111 Line too long in DIRECT text file
- 1112 Decode error in text file READ BOOLEAN
- 1113 Value out of range in text file READ CHAR
- 1114 Decode error in text file READ INTEGER
- 1115 Decode error in text file READ SINT (integer subrange)
- 1116 Decode error in text file READ REAL
- 1117 LSTRING target not big enough in READSET

- 1118 Decode error in text file READ WORD
- 1119 Decode error in text file READ BYTE (word subrange)
- 1120 SEEK call to file not in DIRECT mode
- 1121 SEEK call to file not in DIRECT mode
- 1122 Encode error (field width >255) in text file WRITE BOOLEAN
- 1123 Encode error (field width >255) in text file WRITE INTEGER
- 1124 Encode error (field width >255) in text file WRITE REAL
- 1125 Encode error (field width >255) in text file WRITE WORD
- 1126 Decode error (field width >255) in text file READ INTEGER4
- 1127 Encode error (field width >255) in text file READ INTEGER4
- 1127 Encode error (field width >255) in text file WRITE INTEGER4

H.5 OTHER RUNTIME ERRORS (2000-2999)

Nonfile system error codes range from 2000 to 2999. In some cases, metacommands control whether or not the compiler checks for the error. In other cases, the compiler always checks. The list below indicates which, if any, metacommand controls the error checking.

H.5.1 Memory Errors (2000-2049)

\$RNFILC Illegal command: ".Errors>memory"

on output page H-37; on input line 15747 of page 1 of file "DS"

Since the stack and the heap grow toward each other, all memory errors are related; for example, a stack overflow can cause a "Heap Is Invalid" error if \$STACKCK is off and the stack overflows.

2000 Stack Overflow

The stack (frame) ran out of memory while calling a procedure or function. The compiler checks this condition if the \$STACKCK metacommand is on, and may check it in some other cases.

2001 No Room In Heap

The heap ran out room for a new variable during the NEW (GETHQQ) procedure. The compiler always detects this error.

Error Messages

2002 Heap Is Invalid

During the NEW (GETHQQ) procedure, the allocation algorithm discovered the heap structure is wrong. The compiler always detects this error.

2004 Allocation Internal Error

There was an unexpected error return when GETHQQ was requesting additional heap space from the operating system. Please report occurrences of this error to the Wang Professional Computer Assistance Center at the number listed on Page H-3.

2031 NIL Pointer Reference

DISPOSE or \$NILCK+ found an uninitialized (value 1) pointer. This only occurs if the metacommand \$INITCK is on.

2033 Invalid Pointer Range

DISPOSE OR \$NILCK+ found a pointer that does not point into the heap or is otherwise invalid. (It may have pointed to a disposed block removed from the heap and given back to the system.)

2034 Pointer To Disposed Var

DISPOSE or \$NILCK+ found a pointer to a heap block that has been disposed. Calling DISPOSE twice for the same variable is invalid.

2035 Long DISPOSE Sizes Unequal

In a long form of DISPOSE, the actual length of the variable did not equal the length based on the tag values given.

H.5.2 Ordinal Arithmetic Errors (20500-2009)**2050 No CASE Value Matches Selector**

In a CASE statement without an OTHERWISE clause, none of the branch statements had a CASE constant value equal to the selector expression value. The compiler checks for this error if the \$RANGECK metacommand is on.

2051 Unsigned Divide By Zero

A WORD value was divided by zero. This error is checked only if the \$MATHCK metacommand is on.

2052 Signed Divide by Zero

An INTEGER value was divided by zero. The compiler checks for this error only if the \$MATHCK metaccommand is on.

2053 Unsigned Math Overflow

A WORD result is outside the range zero to MAXWORD. The compiler checks for this error only if the \$MATHCK metaccommand is on.

2054 Signed Math Overflow

An INTEGER result is outside the range from -MAXINT to -MAXINT. The compiler checks for this error only if the \$MATHCK metaccommand is on.

2055 Unsigned Value Out of Range

The source value for assignment or value parameter is out of range for the target value. The target can be a subrange of WORD (including BYTE), or CHAR, or an enumerated type. This error can also occur in SUCC and PRED functions and when you assign the length 1 space only of an LSTRING. The compiler checks for all of these conditions if the \$RANGECK metaccommand is on.

The error also occurs when an array index is out of bounds and the array has an unsigned index type. The compiler checks for this condition when the \$INDEXCK metaccommand is on.

2056 Signed Value Out of Range

This error is similar to #2055, but applies to the INTEGER type and its subranges.

2057 Uninitialized 16 Bit Integer Used

Either an INTEGER or 16-bit INTEGER subrange variable appears without being assigned first, or such a variable has the invalid value of -32768. The compiler checks for this condition if the \$INITCK metaccommand is on.

2058 Uninitialized 8 Bit Integer Used

Either a SINT or 8-bit INTEGER subrange variable appears without being assigned first, or such a variable has the invalid value of -128. The compiler checks for this condition if the 4initck metaccommand is on.

2084 Integer Zero To Negative Power

There was an attempt to raise zero to a negative power (FORTRAN error only).

H.5.3 Type REAL Arithmetic Errors (2100-2149)

2100 REAL Divide By Zero

A REAL value is divided by zero. The compiler always detects this error.

2101 REAL Math Overflow

A REAL value is too large for representation. The compiler always detects this error.

2102 SIN or COS Argument Range

The parameter for a SIN or COS function is too large to yield a meaningful result. This error is only caught in 8080 systems.

2103 EXP Argument Range

The parameter for an EXP function is too large to yield a result that fits in representation. This error is only caught in 8080 systems.

2104 SQRT of Negative Argument

The parameter for a square root function is less than zero. The compiler always detects this error.

2105 LN of Non-Positive Argument

The parameter of a natural log function is less than or equal to zero. The compiler always detects this error.

2106 TRUNC/ROUND Argument Range

The REAL parameter of a TRUNC, TRUNC4, ROUND, or ROUND4 function is outside the range of INTERGERS. The compiler always detects this error.

2131 Tangent Argument Too Small

The parameter for a TANRQQ or ASCRQQ function is greater than one. The compiler always detects this error.

2132 Arcsin or Arccos of REAL > 1.0

The parameter of an ASNRQQ or ACSRQQ function is greater than one. The compiler always detects this error.

2133 Negative Real To Real Power

The first argument of an PRDRQQ or PRSRQQ function is less than zero. The compiler always detects this error.

2134 Real Zero To Negative Power

There was an attempt to raise zero to a negative power in one of the functions PISRQQ, PIDRQQ, PRDRQQ, or PRSRQQ.

2135 REAL Math Underflow

You reduced the significance of a REAL expression to zero.

2136 REAL Infinity (Uninitialized Or Previous Error)

The REAL value called "infinity" was encountered. This may occur if the \$INITCK metaccommand is on and an unitialized REAL value is used, or if a previous error set a variable to indefinite as part of its masked error response.

2137 Missing Arithmetic Processor

You linked your program with the runtime library intended for use with the 8087 numeric coprocessor, but there is no coprocessor on your system. Relink your program with the runtime library that emulates floating-point arithmetic.

2138 REAL IEEE Denormal Detected

A very small real number was generated and may no longer be valid due to loss of significance.

H.5.4 Structured Errors (2150-2199)

2150 String Too Long in COPYSTR

The source string for a COPYSTR intrinsic function was too large for the target string. The compiler always detects this error.

2151 Lstring Too Long In Intrinsic Procedure

The target LSTRING was too small in an INSERT, DELETE, CONCAT, or COPYLST intrinsic procedure. The compiler always detects this error.

2180 Set Element Greater Than 255

The value in a constructed set exceeds the maximum of 255. The compiler always detects this error.

2181 Set Element Out Of Range

The value in a set assignment or set value parameter was too large for the target set. The compiler detects this error only if the \$RANGECK metaccommand is on.

H.5.5 INTEGER4 Errors (2200-2249)

2200 Long Integer Divide By Zero

In INTEGER4 value was divided by zero. The compiler always detects this error.

2201 Long Integer Math Overflow

An INTEGER4 value was too large for representation. The compiler always detects this error.

2234 Long Integer Zero To Negative Power

There was an attempt to raise zero to a negative power (FORTRAN error only).

H.5.6 Other Errors (2400-2999)

2400 Illegal Code

This was an internal error, which may occur in P-code systems only.

2450 Unit Version Number Mismatch

During unit initialization, the user (one with the USES clause) and implementation of an interface were discovered to have been compiled with unequal interface version number. The compiler always detects this error.

Glossary

GLOSSARY

This glossary is limited to definitions of terms associated with the Wang PC Pascal compiler and the compiling process. For a glossary of general terms associated with the Wang Professional Computer, refer to The Wang Professional Computer Introductory Guide.

Compilation

This is the process during which the compiler is compiling a Pascal source file and creating a relocatable object file.

External Reference

This is a variable name or label in a module no other routine in that module references. The Linker tries to resolve this situation by searching for this reference in other modules. If it finds the reference in a module, the Linker loads that module into memory. If it finds the variable or label, it substitutes the address associated with it for the reference in the first module, and the variable is "bound." A variable that is not found is "undefined" or "unresolved."

Linking

This is the process in which the Linker computes absolute addresses for labels and variables in relocatable modules and then resolves all references by searching the runtime library. After loading and linking, the Linker saves the modules that it loaded into memory as a single .EXE file on your disk.

Module

This is a fundamental unit of code, or a series of instructions. Several types of modules exist, including relocatable and executable modules. The compiler creates relocatable modules that the Linker later manipulates. Your final executable program is an example of an executable module.

Relocatable

A module is relocatable if the compiler can run the code within it at different locations in memory. Relocatable modules contain labels called offsets that specify the address of the code in relation to the start of the module. These labels and variables are "code-relative." When the Linker loads the module, it associates an address with the start of the module. The Linker then computes an absolute address that is equal to the associated address plus the code-relative offset for each label or variable. These new values become the absolute addresses used in the binary .EXE file. Library files and .OBJ files are all relocatable modules. Normally, a relocatable module contains global references as well; the compiler resolves these references after all other relocatable modules compute their local labels and variables.

Routine

This is a unit of executable code that resides in a module. A module can contain more than one routine.

Runtime

This is the time during which a compiled and linked program executes. By convention, runtime refers to the execution time of your program and not to the execution time of the compiler or the Linker.

Runtime Library

This contains the routines you need to implement the Wang PC Pascal language. These routines correspond to various features of the Wang PC Pascal language.

Source

Also called a source file, this is a Pascal program that is the input file to the compiler. This file must be in ASCII format.

Index

INDEX

A

ABORT, 23-9
ABS, 23-9
ACDRQQ, 23-9
ACSRQQ, 23-9
AIDRQQ, 23-9
AISRQQ, 23-9
ALLHQ, 23-9, 23-10
ANDRQQ, 23-10
ANSRQQ, 23-10
ARCTAN, 23-10
Array index switch - /A,
 4-9, 26-18
Arrays, 15-1, 15-2
ASCII character set, D-1, D-2
ASCII structure files, 16-4
ASDRQQ, 23-10
ASSIGN, 23-10
ASSRQQ, 23-10
ATDRQQ, 23-10
ATSRQQ, 23-10
Attributes, 19-7
 EXTERN, 19-9
 FORTRAN, 22-9
 INTERRUPT, 22-10
 ORIGIN, 19-10, 22-9
 PORT, 19-10
 PUBLIC, 19-9, 22-8
 PURE, 22-11
 READONLY, 19-10, 19-11
 STATIC, 19-8
A2DRQQ, 23-10
A2SRQQ, 23-10

B

Base name, 4-3
Batch command files, 6-1, 6-2
BEGOQQ, 23-11
BEGXQQ, 23-11
BINARY structure files, 16-3
BOOLEAN types, 14-3
BRAVE metaccommand, 26-5
BYLONG, 23-11
BYWORD, 23-11

C

CHAR types, 14-2
CHDRQQ, 23-12
CHR, 23-12
CHSRQQ, 23-12
CLOSE, 23-12
CNDRQQ, 23-12
CNSRQQ, 23-12
Code-relative labels, Glossary-2
Compilation,
 2-3, 4-1 to 4-10,
 Glossary-1
Compiler structure, 9-1 to 9-5
CONCAT, 23-12
Constants, 18-2
 INTEGER, 18-4
 INTEGER4, 18-4
 REAL, 18-3
 WORD, 18-4
COPYLST, 23-12
COPYSTR, 23-12
COS, 23-12

D

DATE, 23-13
Debug code switch - /D,
 4-9, 26-18
Debug code switch - /Q,
 4-9, 26-18
DEBUG metaccommand, 26-5
DECODE, 23-12
Default drive, 4-3
Default file specifications, 4-4
DELETE, 23-13
Demonstration run, 3-1 to 3-5
DIRECT mode files, 16-5
Directives, 22-7
 EXTERN, 22-7
 FORWARD, 22-7
DISBIN, 23-13
DISCARD, 23-13
Diskette contents, 1-2
DISPOSE, 23-13
Drive designations, 4-3

INDEX (continued)

E

ENABIN, 23-14
 ENCODE, 23-14
 ENDOQQ, 23-15
 ENDXQQ, 23-15
 Entering the compiler, 4-6
 specifying all parameters, 4-7
 specifying no parameters,
 4-6 to 4-7
 specifying some parameters, 4-8
 ENTRY metaccommand, 26-6
 EOF, 23-15, 24-4
 BOLN, 23-15, 24-4
 Error Messages, H-1 to H-35
 ERRORS metaccommand, 26-6
 EVAL, 23-15
 EXDRQQ, 23-15
 EXP, 23-16
 EXSRQQ, 23-15
 EXTEND metaccommand, 26-3
 EXTERN attribute, 19-9
 EXTERN directive, 22-7

F

File names, 4-3
 Files, 16-1 to 16-6
 ASCII structure, 16-4
 BINARY structure, 16-3
 DIRECT mode, 16-5
 SEQUENTIAL mode, 16-5
 TERMINAL mode, 16-5
 FILLC, 23-16
 FILLSC, 23-16
 FLOAT, 23-16
 FLOAT4, 23-16
 FORTRAN attribute, 22-9
 FORWARD directive, 22-7
 FREECT, 23-16
 Functions,
 10-4, 10-5, 23-1 to 23-33,
 F-1 to F-3

G

GET, 23-17, 24-2
 Global reference, Glossary-2
 unbound, Glossary-1
 GOTO metaccommand, 26-6
 GIYUQQ, 23-17

H

HIBYTE, 23-17
 HIWORD, 23-17

I

Identifiers, 12-1 to 12-5
 IF THEN ELSE metaccommand, 26-10
 INCLUDE metaccommand, 26-10
 INCONST metaccommand, 26-11
 INDEXCK metaccommand, 26-6
 INITCK metaccommand, 26-6, 26-7
 INPUT, 16-6
 INSERT, 23-17
 INTEGER constants, 18-4
 INTEGER metaccommand, 26-3
 INTEGER types, 14-1
 INTEGER4 constants, 18-4
 INTEGER4 types, 14-7
 Intermediate files, 4-2, 4-3
 INTERRUPT attribute, 22-10

L

LADDOK, 23-18
 Lazy evaluation, 24-6
 LDDRQQ, 23-18
 LDSRQQ, 23-18
 Libraries, 5-3, 5-4
 LINE metaccommand, 26-7
 Line number call switch - /L,
 4-9, 26-18
 LINESIZE metaccommand, 26-12
 Linker listing file, 5-4
 Linker switches, 5-5
 Linking, 5-1 to 5-6, Glossary-1
 LIST metaccommand, 26-12
 LMULOK, 23-18
 LN, 23-18
 LNDRQQ, 23-18
 LNSRQQ, 23-18
 LOBYTE, 23-18
 LOCKED, 23-19
 LOWER, 23-19
 LOWORD, 23-19
 LSTRING, 15-6 to 15-10

INDEX (continued)

M

MARKAS, 23-19
 Math check switch - /M,
 4-9, 26-18
 MATHCK metaccommand, 26-7
 MDDRQQ, 23-20
 MDSRQQ, 23-20
 MEMAVL, 23-20
 Memory limits, 7-1 to 7-9
 MESSAGE, 26-11
 Metaccommands, 10-1, 26-1 to 26-14
 \$BRAVE, 26-5
 \$DEBUG, 26-5
 \$ENTRY, 26-6
 \$ERRORS, 26-6
 \$EXTEND, 26-3
 \$GOTO, 26-6
 \$IF THEN ELSE, 26-10
 \$INCLUDE, 26-10
 \$INCONST, 26-11
 \$INDEXCK, 26-6
 \$INITCK, 26-6, 26-7
 \$INTEGER, 26-3
 \$LINE, 26-7
 \$LINESIZE, 26-12
 \$LIST, 26-12
 \$MATHCK, 26-7
 \$MESSAGE, 26-11
 \$NILCK, 26-7, 26-8
 \$OCODE, 26-13
 \$PAGE, 26-13
 \$PAGEIF, 26-13
 \$PAGESIZE, 26-13
 \$POP, 26-11
 \$PUSH, 26-11
 \$RANGECK, 26-8
 \$REAL, 26-3
 \$ROM, 26-3
 \$RUNTIME, 26-8
 \$SIMPLE, 26-3
 \$SIZE, 26-3
 \$SKIP, 26-13
 \$\$STACKCK, 26-8
 \$\$STANDARD, 26-3
 \$\$SUBTITLE, 26-13
 \$\$SYSTAB, 26-13, 26-14
 \$\$SYSTEM, 26-3
 \$TAGCK, 26-8
 \$TITLE, 26-14
 \$WARN, 26-8

MNDRQQ, 23-20
 MNSRQQ, 23-20
 Module, 10-5, Glossary-1
 MOVEVL, 23-20
 MOVER, 23-21
 MOVESL, 23-21
 MOVESR, 23-21
 MXDRQQ, 23-22
 MXSRQQ, 23-22

N

NEW, 23-22, 23-23
 NILCK metaccommand, 26-7, 26-8
 Notation, 1-3, 10-11,
 11-1 to 11-6

O

Object file, 4-1
 Object listing file, 4-2
 Object modules, 5-1, 5-2
 OCODE metaccommand, 26-13
 ODD, 23-23
 ORD, 23-23, 23-24
 ORIGIN attribute, 19-10, 22-9
 OUTPUT, 16-6

P

PACK, 23-24
 PAGE metaccommand, 26-13
 PAGE procedure, 23-24, 24-5
 PAGEIF metaccommand, 26-13
 PAGESIZE metaccommand, 26-13
 Pascal, resources for learning,
 1-3
 PIDRQQ, 23-24
 PISRQQ, 23-24
 PLYUQQ, 23-25
 Pointer value switch - /N,
 4-9, 26-18
 POP metaccommand, 26-11
 PORT attribute, 19-10
 POSITN, 23-25
 PRDRQQ, 23-25
 PRED, 23-25
 PRSRQQ, 23-25
 Procedure entry switch - /E,
 4-9, 26-18

INDEX (continued)

Procedures, 10-4, 10-5,
23-1 to 23-23, F-1 to F-3
Program development process,
2-2 to 2-5
PTYUQQ, 23-25
PUBLIC attribute, 19-9, 22-8
PURE attribute, 22-11
PUSH metacommmand, 26-11
PUT, 23-25, 24-2

R
RANGECK metacommmand, 26-8
READ, 23-26, 24-10, 24-11
READFN, 23-26
READLN, 23-26, 24-10, 24-11
READONLY attribute, 19-10, 19-11
REA SET, 23-26
REAL constants, 18-3
REAL metacommmand, 26-3
REAL types, 14-5, 14-6
RELEAS, 23-26
Relocatable, Glossary-2
RESET, 23-27, 24-3
RESULT, 23-27
RETYPE, 23-27, 23-28
REWRITE, 23-28, 24-3
ROM metacommmand, 26-3
ROUND, 23-28
ROUND4, 23-28
Routine, Glossary-2
Run file, 5-4
Running a program, 3-5
Runtime, Glossary-2
Runtime architecture, 9-8 to 9-20
Runtime library, Glossary-2
RUNTIME metacommmand, 26-8

S
SADDOK, 23-28
SCANEQ, 23-29
SCANNE, 23-29
SEEK, 23-29
SEQUENTIAL mode files, 16-5
SHDRQQ, 23-29
SHSRQQ, 23-29
SIMPLE metacommmand, 26-3
SIN, 23-29
SIZE metacommmand, 26-3
SIZEOF, 23-29

SKIP metacommmand, 26-13
SMULOK, 23-30
Source file, Glossary-2
Source listing file, 4-1, 4-2
SPEED metacommmand, 26-3
SQR, 23-30
SQRT, 23-30
Stack overflow check switch - /S,
4-9, 26-18
STACKCK metacommmand, 26-8
STANDARD metacommmand, 26-3
STATIC attribute, 19-8
STRING, 15-5
Subrange check switch - /R,
4-9, 26-18
SUBTITLE metacommmand, 26-13
SUCC, 23-30
Suggested disk setup, 2-2
Super arrays, 15-2 to 15-4
Switches, 4-9, 26-18
/A, 4-9, 26-18
/D, 4-9, 26-18
/E, 4-9, 26-18
/L, 4-9, 26-18
/I, 4-9, 26-18
/M, 4-9, 26-18
/N, 4-9, 26-18
/Q, 4-9, 26-18
/R, 4-9, 26-18
/S, 4-9, 26-18
/T, 4-9, 26-18
SYMTAB metacommmand, 26-13, 26-14
SYSTEM metacommmand, 26-3
System requirements, 1-1

T
Tag check switch - /T, 4-9, 26-18
TAGCK metacommmand, 26-8
TERMINAL mode files, 16-5
THDRQQ, 23-30
THSRQQ, 23-30
TICS, 23-30
TIME, 23-31
TITLE metacommmand, 26-14
TNDRQQ, 23-31
TNSRQQ, 23-31
TRUNC, 23-31
TRUNC4, 23-31

INDEX (continued)

Types, 14-1 to 14-7

- BOOLEAN, 14-3
- CHAR, 14-2
- INTEGER, 14-1
- INTEGER4, 14-7
- REAL, 14-5, 14-6
- WORD, 14-2

U

- UADDOK, 23-31
- UMULOK, 23-32
- UNLOCK, 23-32
- UNPACK, 23-32
- Unbound global reference,
 Glossary-1
- UPPER, 23-32

V

- Value check switch - /I,
 4-9, 26-18
- VECTIN, 23-33

W

- Wang PC Pascal, compared to:
 - ISO standard, B-1 to B-4
 - UCSD Pascal, C-3, C-4
- WARN metaccommand, 26-8
- WORD constants, 18-4
- WORD types, 14-2
- WRD, 23-33
- WRITE, 23-33
- WRITELN, 23-33

